W3C Editor's Draft

OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax

W3C Editor's Draft 02 December 2008

This version:
http://www.w3.org/2007/OWL/draft/ED-owl2-syntax-20081202/
Latest editor's draft:
http://www.w3.org/2007/OWL/draft/owl2-syntax/
Previous version:
http://www.w3.org/2007/OWL/draft/ED-owl2-syntax-20081128/ (color-coded diff)

Editors:
Boris Motik, Oxford University
Peter F. Patel-Schneider, Bell Labs Research, Alcatel-Lucent
Bijan Parsia, University of Manchester
Contributors:
Conrad Bock, National Institute of Standards and Technology (NIST)
Achille Fokoue, IBM Corporation
Peter Haase, Forschungszentrum Informatik (FZI)
Rinke Hoekstra, University of Amsterdam
Ian Horrocks, Oxford University
Alan Ruttenberg, Science Commons (Creative Commons)
Uli Sattler, University of Manchester
Mike Smith, Clark & Parsia

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

Abstract

OWL 2 extends the W3C OWL Web Ontology Language with a small but useful set of features that have been requested by users, for which effective reasoning algorithms are now available, and that OWL tool developers are willing to support. The new features include extra syntactic sugar, additional property and qualified

cardinality constructors, extended datatype support, simple metamodeling, and extended annotations.
This document defines OWL 2 ontologies in terms of their structure, and it also defines a functional-style syntax in which ontologies can be written. Furthermore, this document provides an informal description of each of the constructs provided by the language.

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May Be Superseded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/.

Set of Documents

This document is being published as one of a set of 11 documents:

1. Structural Specification and Functional-Style Syntax (this document)
2. Direct Semantics
3. RDF-Based Semantics
4. Conformance and Test Cases
5. Mapping to RDF Graphs
6. XML Serialization
7. Profiles
8. Quick Reference Guide
9. New Features and Rationale
10. Manchester Syntax
11. rdf:text: A Datatype for Internationalized Text

Summary of Changes

This document contains a few changes since the previous version of 08 October 2008.

- The structure of the annotation subsystem has been considerably refactored and extended.
- Conjunction and disjunction of data ranges have been added.
- The usage of *owl:topDataProperty* has been restricted to allow the datatype map to be extensible.
- The formal definition of the datatype map has been slightly changed to make it compatible with the usual W3C definitions.
- The *owl:rational* and *rdf:XMLLiteral* datatypes were added to the datatype map of OWL 2.
- The status of metamodeling has been clarified.
- Certain parts of the document (mainly in Sections 2, 3, and 5) have been rewritten for clarity.

[Show Short TOC]

Contents

W3C Editor's Draft

1 Introduction

This document defines the OWL 2 language. The core part of this specification — called the *structural specification* — is independent of the concrete exchange syntaxes for OWL 2 ontologies. It describes the conceptual structure of OWL 2 ontologies and thus provides a normative abstract model for all (normative and nonnormative) syntaxes of OWL 2. This allows for a clear separation of the essential features of the language from issues related to any particular syntax. Furthermore, such a structural specification of OWL 2 provides the foundation for the implementation of OWL 2 tools such as APIs and reasoners.

This document also defines the *functional-style syntax*, which closely follows the structural specification and allows OWL 2 ontologies to be written in a compact form. This syntax is used in the definitions of the semantics of OWL 2 ontologies, the mappings from and into the RDF/XML exchange syntax, and the different profiles of OWL 2. Concrete syntaxes, such as the functional-style syntax, often provide features not found in the structural specification, such as a mechanism for abbreviating long IRIs.

An OWL 2 ontology is a formal conceptualization of a domain of interest. OWL 2 ontologies consist of the following three different syntactic categories:

- *Entities*, such as classes, properties, and individuals, are identified by IRIs and can be thought of as primitive *terms* or names. Entities represent basic elements of the domain being modeled. For example, a class *a:Person* can be used to model the set of all people. Similarly, the object property *a:parentOf* can be used to model the parent-child relationship. Finally, the individual *a:Peter* can be used to represent a particular person called `"Peter"`.
- *Expressions* represent complex notions in the domain being modeled. For example, a *class expression* describes a set of individuals in terms of the restrictions on the individuals' features.
- *Axioms* are statements that are asserted to be true in the domain being modeled. For example, using a *subclass axiom*, one can state that the class *a:Student* is a subclass of the class *a:Person*.

These three syntactic categories are used to express the *logical* part of OWL 2 ontologies — that is, they are interpreted under a precisely defined semantics that allows useful inferences to be drawn. For example, if an individual *a:Peter* is an instance of the class *a:Student*, and *a:Student* is a subclass of *a:Person*, then from the OWL 2 semantics one can derive that *a:Peter* is also an instance of *a:Person*.

In addition, entities, axioms, and ontologies can be *annotated* in OWL 2. For example, a class can be given a human-readable label that provides a more descriptive name for the class. Annotations have no effect on the logical aspects of an ontology — that is, for the purposes of the OWL 2 semantics, annotations are treated as not being present. Instead, the use of annotations is left to the applications that use OWL 2. For example, a graphical user interface might choose to visualize a class using one of its labels.

Finally, OWL 2 provides basic support for ontology modularization. In particular, an OWL 2 ontology *O* can import another OWL 2 ontology *O'* and thus gain access to all entities, expressions, and axioms in *O'*.

Hide Structural Diagrams Hide Functional-Style Syntax Grammar
Hide Examples

This document defines the structural specification of OWL 2, the functional syntax for OWL 2, and the behavior of datatype maps. Only the parts of the document related to these three purposes are normative. The examples in this document are informative and any part of the document that is specifically identified as informative is not normative. Finally, the informal descriptions of the semantics of OWL 2 constructs in this document are informative; the semantics is precisely specified in a separate document [*OWL 2 Direct Semantics*].

The italicized keywords *must*, *must not*, *should*, *should not*, and *may* specify certain aspects of the normative behavior of OWL 2 tools, and are interpreted as specified in RFC 2119 [*RFC 2119*].

2 Preliminary Definitions

This section presents certain preliminary definitions that are used in the rest of this document.

2.1 Structural Specification

The structural specification of OWL 2 consists of all the figures in this document and the notion of structural equivalence given below. It is used throughout this document to precisely specify the structure of OWL 2 ontologies and the observable behavior of OWL 2 tools. An OWL 2 tool *may* base its APIs and/or internal storage model on the structural specification; however, it *may* also choose a completely different approach as long as its observable behavior conforms to the one specified in this document.

The structural specification is defined using the Unified Modeling Language (UML) [UML], and the notation used is compatible with the Meta-Object Facility (MOF) [MOF]. This document uses only a very simple form of UML class diagrams that are expected to be easily understandable by readers familiar with the basic concepts of object-oriented systems. The names of abstract classes (i.e., classes that are not intended to be instantiated) are written in italic.

Elements of the structural specification are connected by associations, many of which are of the one-to-many type. Associations whose name is preceded by `/` are *derived* — that is, their value is determined based on the value of other associations and attributes. Whether the elements participating in associations are ordered and whether repetitions are allowed is made clear by the following standard UML conventions:

- By default, all associations are sets; that is, the elements in them are unordered and repetitions are disallowed.
- The `{ ordered,nonunique }` attribute is placed next to the association ends that are ordered and in which repetitions are allowed. Such associations have the semantics of lists.

Whether two elements of the structural specification are considered to be the same is captured by the notion of *structural equivalence*, defined as follows. Elements o_1 and o_2 are structurally equivalent if and only if the following conditions hold:

- If o_1 and o_2 are atomic values, such as strings, integers, or IRIs, they are structurally equivalent if they are identical according to the notion of identity specified by the respective atomic type.
- If o_1 and o_2 are unordered associations without repetitions, they are structurally equivalent if each element of o_1 is structurally equivalent to some element of o_2 and vice versa.
- If o_1 and o_2 are ordered associations with repetitions, they are structurally equivalent if they contain the same number of elements and each element of o_1 is structurally equivalent to the element of o_2 with the same index.
- If o_1 and o_2 are complex elements consisting of other elements, they are structurally equivalent if
    - both o_1 and o_2 are of the same type,
    - each element of o_1 is structurally equivalent to the corresponding element of o_2, and
    - each association of o_1 is structurally equivalent to the corresponding association of o_2.

Note that structural equivalence is not a semantic notion, as it is based only on comparing structures.

Example:

The class expression `UnionOf( a:Person a:Animal )` is structurally equivalent to the class expression `UnionOf( a:Animal a:Person )` because the order of the elements in an unordered association is not important.

The class expression `UnionOf( a:Person ComplementOf( a:Person ) )` is not structurally equivalent to *owl:Thing* even though the two expressions are semantically equivalent.

Although set associations are widely used in the specification, sets written in one of the linear syntaxes (e.g., XML or RDF/XML) are not necessarily expected to be duplicate free. Duplicates *should* be eliminated from such constructs during parsing.

Example:

An ontology written in functional-style syntax can contain a class expression of the form `UnionOf( a:Person a:Animal a:Animal )`. During parsing, this expression should be "flattened" to give the expression `UnionOf( a:Person a:Animal )`.

2.2 BNF Notation

Grammars in this document are specified using the standard BNF notation, summarized in Table 1.

Table 1. The BNF Notation

Construct	Syntax	Example
nonterminal symbols	boldface	**ClassExpression**
terminal symbols	single quoted	`'PropertyRange'`
zero or more	curly braces	{ **ClassExpression** }
zero or one	square brackets	[**ClassExpression**]
alternative	vertical bar	**Assertion** | **Declaration**

Terminal symbols used in the **full-IRI**, **irelative-ref**, **NCName**, **languageTag**, **nodeID**, **nonNegativeInteger**, and **quotedString** productions are defined by specifying their structure in English; to stress this, the English description is italicized.

Whitespace is a maximal sequence of space (U+20), horizontal tab (U+9), line feed (U+A), and carriage return (U+D) characters not occurring within a pair of `"` characters (U+22). A *comment* is a maximal sequence of characters that starts with the `#` (U+23) character and contains neither a line feed (U+A) nor a carriage return (U+D) character.

Whitespace and comments cannot occur within terminal symbols of the grammar. Whitespace and comments can occur between any two terminal symbols of the grammar, and all whitespace *must* be ignored. Whitespace *must* be introduced between a pair of terminal symbols if each terminal symbol in the pair consists solely of alphanumeric characters or matches the **full-IRI**, **irelative-ref**, **NCName**, **nodeID**, or **quotedString** production.

2.3 IRIs and Namespaces

Ontologies and their elements are identified using International Resource Identifiers (IRIs) [RFC3987]; thus, OWL 2 extends OWL 1, which uses Uniform Resource Identifiers (URIs). In the structural specification, IRIs are represented by the **IRI** class. All IRIs in this specification are written using the grammar described below.

An IRI can be written as a full IRI. The < (U+3C) and > (U+3E) characters surrounding a full IRI are not part of the IRI, but are used solely for quotation purposes, identifying an IRI as a full IRI.

Alternatively, an IRI it can be abbreviated as a CURIE [CURIE]. To this end, commonly used IRIs — called *namespaces* — are associated with a *prefix*. An IRI *I* *belongs* to a namespace *NI* if, in their string representation, *NI* is a prefix of *I*; the part of *I* not covered by *NI* is called a *reference* of *I* w.r.t. *NI*. An IRI *I* belonging to a namespace *NI* associated with a prefix *pref* is then commonly abbreviated as a CURIE *pref:ref*, where *ref* is the reference of *I* w.r.t. *NI*. CURIEs are not represented in the structural specification of OWL 2: if a concrete syntax of OWL 2 uses CURIEs to abbreviate long IRIs, these abbreviations *must* be expanded into full IRIs during parsing according to the rules of the respective syntax.

```
full-IRI := 'IRI as defined in [RFC3987], enclosed in a pair
of < (U+3C) and > (U+3E) characters'
NCName := 'as defined in [XML Namespaces]'
irelative-ref := 'as defined in [RFC3987]'

namespace := full-IRI
prefix := NCName
reference := irelative-ref
curie := [ [ prefix ] ':' ] reference

IRI := full-IRI | curie
```

Table 2 defines the standard namespaces and the respective prefixes used throughout this specification.

Table 2. Standard Namespaces and Prefixes Used in OWL 2

Namespace prefix	Namespace

rdf	*<http://www.w3.org/1999/02/22-rdf-syntax-ns#>*
rdfs	*<http://www.w3.org/2000/01/rdf-schema#>*
xsd	*<http://www.w3.org/2001/XMLSchema#>*
owl	*<http://www.w3.org/2002/07/owl#>*

IRIs belonging to the *rdf*, *rdfs*, *xsd*, and *owl* namespaces constitute the *reserved vocabulary* of OWL 2. As described in the following sections, the IRIs from the reserved vocabulary that are listed in Table 3 have special treatment in OWL 2. All IRIs from the reserved vocabulary not listed in Table 3 constitute the *disallowed vocabulary* of OWL 2 and *must not* be used in OWL 2 to name entities, ontologies, or ontology versions.

Table 3. Reserved Vocabulary of OWL 2 with Special Treatment

owl:backwardCompatibleWith	*owl:bottomDataProperty*	*owl:bottomObjectProperty*	*owl:dateTime*	*c*
owl:incompatibleWith	*owl:Nothing*	*owl:priorVersion*	*owl:rational*	*c*
owl:realPlus	*owl:Thing*	*owl:topDataProperty*	*owl:topObjectProperty*	*r*
rdf:text	*rdf:XMLLiteral*	*rdfs:comment*	*rdfs:isDefinedBy*	*r*
rdfs:Literal	*rdfs:seeAlso*	*xsd:anyURI*	*xsd:base64Binary*	*x*
xsd:byte	*xsd:decimal*	*xsd:double*	*xsd:float*	*x*
xsd:int	*xsd:integer*	*xsd:language*	*xsd:length*	*x*
xsd:maxExclusive	*xsd:maxInclusive*	*xsd:maxLength*	*xsd:minExclusive*	*x*
xsd:minLength	*xsd:Name*	*xsd:NCName*	*xsd:negativeInteger*	*x*
xsd:nonNegativeInteger	*xsd:nonPositiveInteger*	*xsd:normalizedString*	*xsd:pattern*	*x*
xsd:short	*xsd:string*	*xsd:token*	*xsd:unsignedByte*	*x*
xsd:unsignedLong	*xsd:unsignedShort*			

2.4 Integers, Strings, Language Tags, and Node IDs

Several types of syntactic elements are commonly used in this document. Nonnegative integers are defined as usual.

```
nonNegativeInteger := 'a nonempty finite sequence of digits
between 0 and 9'
```

Characters and strings are defined in the same way as in [*RDF:TEXT*]. A *character* is an atomic unit of communication. The structure of characters is not further specified in OWL 2, other than to note that each character has a Universal Character Set (UCS) code point [*ISO/IEC 10646*]. The set of available characters is assumed to be infinite, and is thus independent from the currently actual version of UCS. A *string* is a finite sequence of characters, and the *length* of a string is the number of characters in it. In this document, strings are written as specified in [*RDF:TEXT*]: they are enclosed in double quotes (U+22), and a subset of the escaping mechanism of the N-triples specification [*RDF Test Cases*] is used to encode strings containing quotes.

quotedString := *'a finite sequence of characters in which " (U+22) and \ (U+5C) occur only in pairs of the form \" (U+22, U+5C) and \\ (U+22, U+22), enclosed in a pair of " (U+22) characters'*

Language tags are nonempty strings as defined in BCP 47 [*BCP 47*]. In this document, language tags are not enclosed in double quotes; however, this does not lead to parsing problems since, according to BCP 47, language tags contain neither whitespace nor the parenthesis characters `(` (U+28) and `)` (U+29).

languageTag := *'a nonempty (not quoted) string defined as specified in BCP 47 [BCP 47]'*

Node IDs are borrowed from the N-Triples specification [*RDF Test Cases*].

nodeID := *'a node ID of the form* `_:name` *as specified in the N-Triples specification [RDF Test Cases]'*

3 Ontologies

An OWL 2 *ontology* is an instance *O* of the **Ontology** class from the structural specification of OWL 2 shown in Figure 1 that satisfies certain conditions given below. The main component of an OWL 2 ontology is its set of axioms, the structure of which is described in more detail in Section 9. Because the association between an ontology and its axioms is a set, an ontology cannot contain two axioms that are structurally equivalent. Apart from the axioms, ontologies can also contain ontology annotations (as described in more detail in Section 3.5), and they can also import other ontologies (as described in Section 3.4).

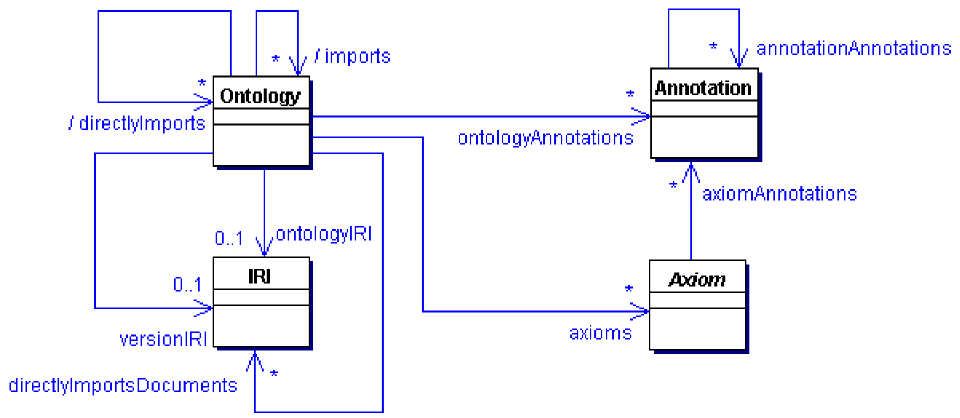

Figure 1. The Structure of OWL 2 Ontologies

The following list summarizes all the conditions that *O* is required to satisfy to be an OWL 2 ontology.

- *O must* satisfy the restrictions on the presence of the ontology IRI and version IRI from Section 3.1;
- *O should* satisfy the constraints on the uniqueness of the ontology IRI and version IRI from Section 3.1;
- *O should* satisfy the restrictions on the import closure from Section 3.4;
- each entity in *O must* have an IRI satisfying the restrictions on the usage of the reserved vocabulary from Sections 5.1–5.6;
- each datatype in *O must* satisfy the restriction from Section 5.2;
- each literal in *O must* satisfy the restriction from Section 5.7;
- *O must* satisfy the typing constraints from Section 5.8.1;
- each **DatatypeRestriction** in *O must* satisfy the restriction from Section 7.5;
- each **DataSomeValuesFrom** and **DataAllValuesFrom** class expression in *O must* satisfy the restrictions from Section 8.4.1 and Section 8.4.2;
- each **DataPropertyRange** axiom in *O must* satisfy the restriction from Section 9.3.5;
- *O must* satisfy the global restriction from Section 11; and
- each *O'* directly imported into *O must* satisfy all of these restrictions as well.

An instance *O* of the **Ontology** class *may* have consistent declarations as specified in Section 5.8.2; however, this is not strictly necessary to make *O* an OWL 2 ontology.

3.1 Ontology IRI and Version IRI

Each ontology *may* have an *ontology IRI*, which is used to identify an ontology. If an ontology has an ontology IRI, the ontology *may* additionally have a *version IRI*, which is used to identify the version of the ontology. The version IRI *may*, but need not be equal to the ontology IRI. An ontology without an ontology IRI *must not* contain a version IRI.

The following list provides conventions for choosing ontology IRI and version IRI in OWL 2 ontologies. This specification provides no mechanism for enforcing these constraints across the entire Web; however, OWL 2 tools *should* use them to detect problems in ontologies they process.

- If an ontology has an ontology IRI but no version IRI, then a different ontology with the same ontology IRI but no version IRI *should not* exist.
- If an ontology has both an ontology IRI and a version IRI, then a different ontology with the same ontology IRI and the same version IRI *should not* exist.
- All other combinations of the ontology IRI and version IRI are not required to be unique. Thus, two different ontologies *may* have no ontology IRI and no version IRI; similarly, an ontology containing only an ontology IRI *may* coexist with another ontology with the same ontology IRI and some other version IRI.

The ontology IRI and the version IRI together identify a particular version from an *ontology series* — the set of all the versions of a particular ontology identified using a common ontology IRI. In each ontology series, exactly one ontology version is regarded as the *current* one. Structurally, a version of a particular ontology is an instance of the **Ontology** class from the structural specification. Ontology series are not represented explicitly in the structural specification of OWL 2—they exist only as a side-effect of the naming conventions described in this and the following sections.

3.2 Ontology Documents

An OWL 2 ontology is an abstract notion defined in terms of the structural specification. Each ontology is associated with an *ontology document*, which physically contains the ontology stored in a particular way. The name "ontology document" reflects the expectation that a large number of ontologies will be stored in physical text documents written in one of the syntaxes of OWL 2. OWL 2 tools, however, are free to devise other types of ontology documents — that is, to introduce other ways of physically storing ontologies.

Ontology documents are not represented in the structural specification of OWL 2, and the specification of OWL 2 makes only the following two assumptions about their nature:

- Each ontology document can be accessed from an IRI by means of an appropriate protocol.
- Each ontology document can be converted in some well-defined way into an ontology (i.e., into an instance of the **Ontology** class from the structural specification).

Example:

An OWL 2 tool might publish an ontology as a text document written in the functional-style syntax (see Section 3.7) and accessible from the IRI <*http://www.example.com/ontology*>. An OWL 2 tool could also devise a scheme for storing OWL 2 ontologies in a relational database. In such a case, each subset of the database representing the information about one ontology corresponds to one ontology document. To provide a mechanism for accessing these ontology documents, the OWL 2 tool should identify different database subsets with distinct IRIs.

The ontology document of an ontology *O should* be accessible from the IRIs determined by the following rules:

- If *O* does not contain an ontology IRI (and, consequently, it does not contain a version IRI either), then the ontology document of *O may* be accessible from any IRI.
- If *O* contains an ontology IRI *OI* but no version IRI, then the ontology document of *O should* be accessible from the IRI *OI*.
- If *D* contains an ontology IRI *OI* and a version IRI *VI*, then the ontology document of *O should* be accessible from the IRI *VI*; furthermore, if *O* is the current version of the ontology series with the IRI *OI*, then the ontology document of *O should* also be accessible from the IRI *OI*.

Thus, the document containing the current version of an ontology series with some IRI *OI should* be accessible from *OI*. To access a particular version of *OI*, one needs to know that version's version IRI *VI*; then, the ontology document *should* be accessible from *VI*.

Example:

An ontology document of an ontology that contains an ontology IRI <*http://www.example.com/my*> but no version IRI should be accessible from the IRI <*http://www.example.com/my*>. In contrast, an ontology document of an ontology that contains an ontology IRI <*http://www.example.com/my*> and a version IRI <*http://www.example.com/my/2.0*> should be accessible from the IRI <*http://www.example.com/my/2.0*>. In both cases, the ontology document should be accessible from the respective IRIs using the HTTP protocol.

OWL 2 tools will often need to implement functionality such as caching or off-line processing, where ontology documents may be stored at addresses different from the ones dictated by their ontology IRIs and version IRIs. OWL 2 tools *may* implement a *redirection* mechanism: when a tool is used to access an ontology document at IRI *I*, the tool *may* redirect *I* to a different IRI *DI* and access the ontology document from there instead. The result of accessing the ontology document from *DI must* be the same as if the ontology were accessed from *I*.

Furthermore, once the ontology document is converted into an ontology, the ontology *should* satisfy the three conditions from the beginning of this section in the same way as if it the ontology document were accessed from *I*. No particular redirection mechanism is specified — this is assumed to be implementation dependent.

Example:

To enable off-line processing, an ontology document that — according to the above rules — should be accessible from <*http://www.example.com/my*> might be stored in a file accessible from <*file:///usr/local/ontologies/example.owl*>. To access this ontology document, an OWL 2 tool might redirect the IRI <*http://www.example.com/my*> and actually access the ontology document from <*file:///usr/local/ontologies/example.owl*>. The ontology obtained after accessing ontology document should satisfy the usual accessibility constraints: if the ontology contains only the ontology IRI, then the ontology IRI should be equal to <*http://www.example.com/my*>, and if the ontology contains both the ontology IRI and the version IRI, then one of them should be equal to <*http://www.example.com/my*>.

3.3 Versioning of OWL 2 Ontologies

The conventions from Section 3.2 provide a simple mechanism for versioning OWL 2 ontologies. An ontology series is identified using an ontology IRI, and each version in the series is assigned a different version IRI. The ontology document of the ontology representing the current version of the series *should* be accessible from the ontology IRI and, if present, at its version IRI as well; the ontology documents of the previous versions *should* be accessible solely from their respective version IRIs. When a new version *O* in the ontology series is created, the ontology document of *O* *should* replace the one acessible from the ontology IRI (and it *should* also be accessible from its version IRI).

Example:

The ontology document containing the current version of an ontology series might be accessible from the IRI <*http://www.example.com/my*>, as well as from the version-specific IRI <*http://www.example.com/my/2.0*>. When a new version is created, the ontology document of the previous version should remain accessible from <*http://www.example.com/my/2.0*>; the ontology document of the new version, called, say, <*http://www.example.com/my/3.0*>, should be made accessible from both <*http://www.example.com/my*> and <*http://www.example.com/my/3.0*>.

3.4 Imports

An OWL 2 ontology can import other ontologies in order to gain access to their entities, expressions, and axioms, thus providing the basic facility for ontology modularization.

Example:

Assume that one wants to describe research projects about diseases. Managing information about the projects and the diseases in the same ontology might be cumbersome. Therefore, one might create a separate ontology *O* about diseases and a separate ontology *O'* about projects. The ontology *O'* would import *O* in order to gain access to the classes representing diseases; this allows one to use the diseases from *O* when writing the axioms of *O'*.

From a physical point of view, an ontology contains a set of IRIs, shown in Figure 1 as the **directlyImportsDocuments** association; these IRIs identify the ontology documents of the directly imported ontologies as specified in Section 3.2. The logical *directly imports* relation between ontologies, shown in Figure 1 as the **directlyImports** association, is obtained by accessing the directly imported ontologies and converting them into OWL 2 ontologies. The logical *imports* relation between ontologies, shown in Figure 1 as the **imports** association, is the transitive closure of directly imports. In Figure 1, associations **directlyImports** and **imports** are shown as derived associations, since their values are derived from the value of the **directlyImportsDocuments** association. Ontology documents usually store the **directlyImportsDocuments** association. In contrast, the **directlyImports** and **imports** associations are typically not stored in ontology documents, but are determined during parsing as specified in Section 3.6.

Example:

The following functional-style syntax ontology document contains an ontology that directly imports an ontology contained in the ontology document accessible from IRI <*http://www.example.com/my/2.0*>.

```
Ontology(<http://www.example.com/importing-ontology>
    Import(<http://www.example.com/my/2.0>)

...
)
```

The IRIs identifying the ontology documents of the directly imported ontologies can be redirected as described in Section 3.2. For example, in order to access the ontology document from a local cache, the ontology document <*http://www.example.com/my/2.0*> might be redirected to <*file:///usr/local/*

ontologies/imported.v20.owl>. Note that this can be done without changing the ontology document of the importing ontology.

The *import closure* of an ontology *O* is a set containing *O* and all the ontologies that *O* imports. The import closure of *O should not* contain ontologies O_1 and O_2 such that

- O_1 and O_2 are different ontology versions from the same ontology series, or
- O_1 contains an ontology annotation *owl:incompatibleWith* with the value equal to either the ontology IRI or the version IRI of O_2.

The *axiom closure* of an ontology *O* is the smallest set that contains all the axioms from each ontology *O'* in the import closure of *O* with all anonymous individuals *renamed apart* — that is, the anonymous individuals from different ontologies in the import closure of *O* are treated as being different; see Section 5.6.2 for further details.

3.5 Ontology Annotations

An OWL 2 ontology contains a set of annotations. These can be used to associate information with an ontology — for example the ontology creator's name. As discussed in more detail in Section 10, each annotation consists of an annotation property and an annotation value, and the latter can be a literal, an IRI, or an anonymous individual. Ontology annotations do not affect the logical meaning of the ontology.

```
ontologyAnnotations := { Annotation }
```

OWL 2 provides several built-in annotation properties for ontology annotations. The usage of these annotation properties on entities other than ontologies is discouraged.

- The *owl:priorVersion* annotation property specifies the IRI of a prior version of the containing ontology.
- The *owl:backwardCompatibleWith* annotation property specifies the IRI of a prior version of the containing ontology that is compatible with the current version of the containing ontology.
- The *owl:incompatibleWith* annotation property specifies the IRI of a prior version of the containing ontology that is incompatible with the current version of the containing ontology.

3.6 Canonical Parsing

Many OWL 2 tools need to support *ontology parsing* — the process of converting an ontology document written in a particular syntax into an OWL 2 ontology. In order to be able to instantiate the appropriate classes from the structural specification, the ontology parser sometimes needs to know which IRIs are used in the ontology as entities of which type. This typing information is extracted from declarations — axioms that associate IRIs with entity types. Please refer to Section 5.8 for more information about declarations.

Example:

An ontology parser for the ontology documents written in functional-style syntax might encounter the following axiom:

```
SubClassOf ( a:Father SomeValuesFrom( a:parentOf
a:Child ) )
```

From this axiom alone, it is not clear whether *a:parentOf* is an object or a data property, and whether *a:Child* is a class or a datatype. In order to disambiguate the types of these IRIs, the parser needs to look at the declarations in the ontology document being parsed, as well as those in the directly or indirectly imported ontology documents.

In OWL 2 there is no requirement for a declaration of an entity to physically precede the entity's usage in ontology documents; furthermore, declarations for entities can be placed in imported ontology documents and imports are allowed to be cyclic. In order to precisely define the result of ontology parsing, this specification defines the notion of *canonical parsing*. An OWL 2 parser *may* implement parsing in any way it chooses, as long as it produces a result that is structurally equivalent to the result of canonical parsing.

An OWL 2 ontology corresponding to an ontology document D_{GI} accessible at a given IRI *GI* can be obtained using the following *canonical parsing* process. All steps of this process *must* be successfully completed.

CP-1 Make *AllDoc* and *Processed* equal to the empty set, and make *ToProcess* equal to the set containing only the IRI *GI*.

CP-2 While *ToProcess* is not empty, remove an arbitrary IRI *I* from it and, if *I* is not contained in *Processed*, perform the following steps:

CP-2.1 Retrieve the ontology document D_I from *I* as specified in Section 3.2.

CP-2.2 Using the rules of the relevant syntax, analyze *D* and compute the set *Decl*(D_I) of declarations explicitly present in D_I and the set *Imp*(D_I) of IRIs of ontology documents directly imported in D_I.

CP-2.3 Add D_I to *AllDoc*, add *I* to *Processed*, and add each IRI from *Imp(D_I)* to *ToProcess*.

CP-3 For each ontology document *D* in *AllDoc*, perform the following steps:

CP-3.1 Compute the set *AllDecl(D)* as the union of the set *Decl(D)*, the sets *Decl(D')* for each ontology document *D'* that is (directly or indirectly) imported into *D*, and the set of all declarations listed in Table 9. The set *AllDecl(D) must* satisfy the typing constraints from Section 5.8.1.

CP-3.2 Create an instance O_D of the **Ontology** class from the structural specification.

CP-3.3 Using the rules of the relevant syntax, analyze *D* and populate O_D by instantiating appropriate classes from the structural specification. Use the declarations in *AllDecl(D)* to disambiguate IRI references if needed; it *must* be possible to disambiguate all IRI references.

CP-4 For each pair of ontology documents *DS* and *DT* in *AllDoc* such that the latter is directly imported into the former, add O_{DT} to the **directlyImports** association of O_{DS}.

CP-5 For each ontology document *D* in *AllDoc*, set the **imports** association of O_D to the transitive closure of the **directlyImports** association of O_D.

CP-6 For each ontology document *D* in *AllDoc*, ensure that O_D is an OWL 2 ontology — that is, O_D *must* satisfy all the restrictions listed in Section 3.

It is important to understand that canonical parsing merely defines the result of the parsing process, and that an implementation of OWL 2 *may* optimize this process in numerous ways. In order to enable efficient parsing, OWL 2 implementations are encouraged to write ontologies into documents by placing all IRI declarations before the axioms that use these IRIs; however, this is not required for conformance.

Example:

A parser for the functional-style syntax of OWL 2 can parse the ontology in a single pass when the declarations for the IRIs are placed in the text of *O* physically before the IRIs are used. Similarly, a parser can optimize the handling of imported ontologies in cases when the import relation between the ontologies is acyclic.

3.7 Functional-Style Syntax

A *functional-style syntax ontology document* is a sequence of Unicode characters [*UNICODE*] accessible from some IRI by means of the standard protocols such that its text matches the **ontologyDocument** production of the grammar defined in this specification document, and it can be converted into an ontology by means of the canonical parsing process described in Section 3.6 and other parts of this specification document. A functional-style syntax ontology document *should* use the UTF-8 encoding [*RFC3629*].

```
ontologyDocument := { prefixDefinition } Ontology
prefixDefinition := 'Namespace' '(' [ prefix ] '=' namespace ')'
Ontology :=
    'Ontology' '(' [ ontologyIRI [ versionIRI ] ]
       directlyImportsDocuments
       ontologyAnnotations
       axioms
    ')'
ontologyIRI := IRI
versionIRI := IRI
directlyImportsDocuments := { 'Import' '(' IRI ')' }
axioms := { Axiom }
```

Example:

The following is a functional-style syntax ontology document containing an ontology with the ontology IRI <*http://www.example.com/ontology1*>. This ontology imports an ontology whose ontology document should be accessed from <*http://www.example.com/ontology2*>, and it contains an ontology annotation providing a label for the ontology and a single subclass axiom.

```
Ontology(<http://www.example.com/ontology1>
    Import(<http://www.example.com/ontology2>)
    Annotation( rdfs:label "The example" )

    SubClassOf( a:Child a:Person )
)
```

Each part of the ontology document matching the **prefixDefinition** production associates a prefix with a namespace. An ontology document *must* contain at most one such definition per prefix and at most one such definition without a prefix, and it *must not* contain a definition for a prefix listed in Table 2. Prefix definitions are used during parsing to expand CURIEs in the ontology document — that is, parts of the ontology document matching the **curie** production — into full IRIs as follows. The full IRI obtained by this expansion *must* be a valid IRI.

- If the prefix of the CURIE is not present, then the ontology document being parsed *must* contain a definition without a prefix. The resulting full

IRI is obtained by concatenating the namespace with the CURIE's reference.

- If the prefix of the CURIE is present, then either Table 2 or the prefix definitions of the ontology document being parsed *must* contain a definition associating the prefix with a namespace. The resulting full IRI is obtained by concatenating the namespace with the CURIE's reference.

A functional-style syntax ontology document *D* can be converted into an OWL 2 ontology by instantiating the canonical parsing process from Section 3.6 as follows:

- In step CP-2.2, *Decl(D)* is obtained from the **Declaration** production described in Section 5.8, and *Imp(D)* is obtained from the **directlyImportsDocuments** production described above.
- In step CP-3.3, *AllDecl(D)* is used to disambiguate the **Class**, **Datatype**, **ObjectProperty**, **DataProperty**, **AnnotationProperty**, and **NamedIndividual** productions of the functional-style syntax grammar.

4 Datatype Maps

OWL 2 ontologies can contain literals and datatypes with built-in semantics, which describe well-known objects such as strings or integers. Each kind of such objects is called a *datatype*, and the set of all supported datatypes is called a *datatype map*. A datatype map is not a syntactic construct , so it is not included in the structural specification of OWL 2. Each datatype in a datatype map is identified by an IRI, and it can be used in OWL 2 ontologies as described in Section 5.2. Each datatype in the datatype map is described by the following components:

- The *value space* is a set determining the set of values of the datatype. Elements of the value space are called *data values*.
- The *lexical space* is a set of strings that can be used to refer to data values. Each member of the lexical space is called a *lexical value*, and it is mapped to a particular data value.
- The *facet space* is a set of pairs of the form ⟨ *F v* ⟩, where *F* is an IRI called a *constraining facet*, and *v* is an arbitrary object called a *value*. Each such pair is mapped to a subset of the value space of the datatype.

The OWL 2 datatype map consists of the datatypes described in the rest of this section, most of which are based on XML Schema Datatypes, version 1.1 [*XML Schema Datatypes*]. The definitions of these datatypes in OWL 2 are largely the same as in XML Schema; however, there are minor differences, all of which are clearly identified. These differences were introduced mainly to align the semantics of OWL 2 datatypes with practical use cases.

OWL 2 tools *may* support datatypes that are not listed in this section. The semantic consequences of OWL 2 ontologies depend exclusively on the set of actually used datatypes [*OWL 2 Direct Semantics*], so supporting datatypes not listed in this section does not affect the consequences of OWL 2 ontologies that do not use these extra datatypes.

4.1 Numbers

OWL 2 provides a rich set of datatypes, listed in Table 4, for representing various kinds of numbers.

Value Spaces. The value spaces of all numeric datatypes are shown in Table 4. The value space of *owl:realPlus* contains the value spaces of all other numeric datatypes. The special values *-0*, *+INF*, *-INF*, and *NaN* are not identical to any number. In particular, *-0* is not a real number and it is not identical to real number zero; to stress this distinction, the real number zero is often called a *positive zero*, written *+0*.

Table 4. Numeric Datatypes and Their Value Spaces

Datatype	Value Space
owl:realPlus	the set of all real numbers extended with four *special values -0* (*negative zero*), *+INF* (*positive infinity*), *-INF* (*negative infinity*), and *NaN* (*not-a-number*)
owl:real	the set of all real numbers
owl:rational	the set of all rational numbers
xsd:double	the four special values *-0*, *+INF*, *-INF*, and *NaN*, plus the set of all real numbers of the form $m \times 2^{e}$ where *m* is an integer whose absolute value is less than 2^{53} and *e* is an integer between -1075 and 970, inclusive
xsd:float	the four special values *-0*, *+INF*, *-INF*, and *NaN*, plus the set of all real numbers of the form $m \times 2^{e}$ where *m* is an integer whose absolute value is less than 2^{24} and *e* is an integer between -149 and 104, inclusive
xsd:decimal	the set of all real numbers of the form $i \times 10^{-n}$ where *i* is an integer and *n* is a nonnegative integer
xsd:integer	the set of all integers
xsd:nonNegativeInteger	the set of all nonnegative integers
xsd:nonPositiveInteger	the set of all negative integers plus (positive) zero
xsd:positiveInteger	the set of all positive integers
xsd:negativeInteger	the set of all negative integers
xsd:long	the set of all integers between -9223372036854775808 and 9223372036854775807, inclusive
xsd:int	the set of all integers between -2147483648 and 2147483647, inclusive
xsd:short	the set of all integers between -32768 and 32767, inclusive
xsd:byte	the set of all integers between -128 and 127, inclusive
xsd:unsignedLong	the set of all integers between 0 and 18446744073709551615, inclusive

xsd:unsignedInt	the set of all integers between 0 and 4294967295, inclusive
xsd:unsignedShort	the set of all integers between 0 and 65535, inclusive
xsd:unsignedByte	the set of all integers between 0 and 255, inclusive

Feature At Risk #1: *owl:rational* support

Note: This feature is "at risk" and may be removed from this specification based on feedback. Please send feedback to public-owl-comments@w3.org.

The *owl:rational* datatype might be removed from OWL 2 if implementation experience reveals problems with supporting this datatype.

Lexical Spaces. Datatypes *owl:realPlus* and *owl:real* do not directly provide any lexical values.

The *owl:rational* datatype supports lexical values defined by the following grammar (whitespace within the grammar *must* be ignored and *must not* be included in the lexical values of *owl:dateTime*, and single quotes are used to introduce terminal symbols):

```
numerator '/' denominator
```

where `numerator` is an integer with the syntax as specified for the *xsd:integer* datatype, and `denominator` is a positive, nonzero integer with the syntax as specified for the *xsd:integer* datatype, not containing the plus sign. Each such lexical value of *owl:rational* is mapped to the rational number obtained by dividing `numerator` by `denominator`.

For each numeric datatype *DT* from XML schema, the lexical values of *DT* are defined as specified in XML Schema Datatypes [XML Schema Datatypes]. Furthermore, each lexical value of *DT* is assigned a data value as specified in XML Schema Datatypes [XML Schema Datatypes].

The lexical values of *owl:rational*, *xsd:decimal*, and the datatypes derived from *xsd:integer* are mapped to arbitrarily large and arbitrarily precise numbers. An OWL 2 implementation *may* support all such lexical values; however, it *must* support at least the following *core* lexical values, which can easily be mapped to the primitive values commonly found in modern implementation platforms:

- All *xsd:float* and *xsd:double* lexical values are core lexical values.
- A lexical value of type *owl:rational* is a core lexical value if its numerator and denominator are in the value space of *xsd:long*.
- A lexical value of type *xsd:decimal* is a core lexical value if its data value is a number with absolute value less than 10^{16} and the representation of the number requires at most 16 digits in total.

- A lexical value for *xsd:integer* or a type derived from *xsd:integer* is a core lexical value if its data value is in the value space of *xsd:long*.

Feature At Risk #2: *xsd:decimal* precision

The new XML Schema spec contains an acknowledged editorial error in the definition of core lexical values for *xsd:decimal*. This document will be updated to state that core decimal lexical values are those that can be expressed with sixteen decimal digits, as is stated here. This document will be updated to use the wording in the XML Schema spec if the change there is made in time.

Please send feedback to public-owl-comments@w3.org.

Equality and Ordering. The facet space of the numeric datatypes are based on the following definitions of equality and ordering.

The equality = is the smallest symmetric relation on the value space of *owl:realPlus* such that all of the following conditions hold:

- *x* = *x* if *x* is a real number, *-0*, *-INF*, or *+INF*; and
- *-0* = *+0*.

Note that *NaN* is not equal to itself; furthermore, even though *-0* is equal to *+0*, it is not identical to it.

Example:

To understand the distinction between identity and equality, consider the following example ontology:

`PropertyAssertion( a:Meg a:numberOfChildren "+0"^^xsd:float )`	The value of *a:numberOfChildren* for *a:Meg* is *+0*.
`PropertyAssertion( a:Meg a:numberOfChildren "-0"^^xsd:float )`	The value of *a:numberOfChildren* for *a:Meg* is *-0*.
`FunctionalProperty( a:numberOfChildren )`	An individual can have at most one value for *a:numberOfChildren*.

The last axiom states that no individual should have more than one distinct value for *a:numberOfChildren*. Even though positive and negative zeros are equal, they are *distinct values*; hence, the first two axioms violate the restriction of the last axiom, which leads to inconsistency.

The ordering < is the smallest relation on the value space of *owl:realPlus* such that all of the following conditions hold:

- *x* < *y* if *x* and *y* are real numbers and *x* is smaller than *y*;
- *-INF* < *x* < *+INF* for each real number *x*;
- *-INF* < *-0* < *+INF*;
- *-0* < *x* for each positive real number *x*; and
- *x* < *-0* for each negative real number *x*.

Note that *+0* is a real number and is thus covered by the first two cases.

Example:

According to the above definition, the subset of the value space of *owl:realPlus* between *-1* and *1* contains both *+0* and *-0*.

Facet Space. The facet space of each numeric datatype *DT* is shown in Table 5.

Table 5. The Facet Space of each Numeric Datatype *DT*

Pair	Facet Value
⟨ *xsd:minInclusive v* ⟩ where *v* is from the value space of *owl:realPlus*	the set of all numbers *x* from the value space of *DT* such that *x* = *v* or *x* > *v*
⟨ *xsd:maxInclusive v* ⟩ where *v* is from the value space of *owl:realPlus*	the set of all numbers *x* from the value space of *DT* such that *x* = *v* or *x* < *v*
⟨ *xsd:minExclusive v* ⟩ where *v* is from the value space of *owl:realPlus*	the set of all numbers *x* from the value space of *DT* such that *x* > *v*
⟨ *xsd:maxExclusive v* ⟩ where *v* is from the value space of *owl:realPlus*	the set of all numbers *x* from the value space of *DT* such that *x* < *v*

Relationship with XML Schema. Numeric datatypes in OWL 2 differ from the numeric datatypes of XML Schema [XML Schema Datatypes] in the following aspects:

- OWL 2 provides the *owl:real*, *owl:realPlus*, and *owl:rational* datatypes.
- The value spaces of all datatypes are subsets of the value space of *owl:realPlus*; thus, unlike in XML Schema, the value spaces of *xsd:float* and *xsd:double* in OWL 2 are not disjoint with the value space of *xsd:decimal*.
- Only a subset of the XML Schema constraining facets are supported.

In other respects, the numeric datatypes of OWL 2 are aligned with the definitions of XML Schema Datatypes [XML Schema Datatypes].

4.2 Strings

OWL 2 uses the *rdf:text* datatype for the representation of strings in a particular language. The definitions of the value space, the lexical space, the facet space, and the necessary mappings are given in [RDF:TEXT].

In addition, OWL 2 supports the following XML Schema Datatypes [XML Schema Datatypes]:

- *xsd:string*
- *xsd:normalizedString*
- *xsd:token*
- *xsd:language*
- *xsd:Name*
- *xsd:NCName*
- *xsd:NMTOKEN*

As recommended in [RDF:TEXT], the value spaces of these datatypes are subsets of the value space of *rdf:text*; please refer to [RDF:TEXT] for a precise definition.

4.3 Boolean Values

The *xsd:boolean* datatype allows for the representation of Boolean values.

Value Space. The value space of *xsd:boolean* is the set containing exactly the two values *true* and *false*. These values are not contained in the value space of any other datatype.

Lexical Space. The *xsd:boolean* datatype supports the following lexical values:

- `"true"` and `"1"` are mapped to the data value *true*, and
- `"false"` and `"0"` are mapped to the data value *false*.

Facet Space. The *xsd:boolean* datatype does not support any constraining facets.

4.4 Binary Data

Datatypes *xsd:hexBinary* and *xsd:base64Binary* allow for the representation of binary data. The two datatypes are the same apart from fact that they support a different syntactic representation for lexical values.

Value Spaces. The value space of both *xsd:hexBinary* and *xsd:base64Binary* is the set of finite sequences of *octets* — integers between 0 and 255, inclusive.

Lexical Spaces. The lexical values of the *xsd:hexBinary* and *xsd:base64Binary* datatypes are strings of the form `"abc"`, whose structure is specified in Sections 3.3.16 and 3.3.17 of XML Schema Datatypes [XML Schema Datatypes], respectively. The lexical values are mapped to data values as specified in XML Schema Datatypes [XML Schema Datatypes].

Facet Space. The facet space of the *xsd:hexBinary* and *xsd:base64Binary* datatypes is shown in Table 6.

Table 6. The Facet Space of the *xsd:hexBinary* and *xsd:base64Binary* Datatypes

Pair	Facet Value
〈 *xsd:minLength v* 〉 where *v* is a nonnegative integer	the set of finite sequences of octets of length at least *v*
〈 *xsd:maxLength v* 〉 where *v* is a nonnegative integer	the set of finite sequences of octets of length at most *v*
〈 *xsd:length v* 〉 where *v* is a nonnegative integer	the set of finite sequences of octets of length exactly *v*

4.5 IRIs

The *xsd:anyURI* datatype allows for the representation of IRIs.

Value Space. The value space of *xsd:anyURI* is the set IRIs as defined in XML Schema Datatypes [XML Schema Datatypes]. Although each IRI has a string representation, the value space of *xsd:anyURI* is disjoint with the value space of *xsd:string*. The string representation of IRIs, however, can be described by a regular expression, so the value space of *xsd:anyURI* is isomorphic to the value space of *xsd:string* restricted with a suitable regular expression.

Lexical Space. The lexical values of the *xsd:anyURI* datatype and their mapping to data values are defined in Section 3.3.18 of XML Schema Datatypes [XML Schema Datatypes].

Note that the lexical values of *xsd:anyURI* include relative IRIs. If an OWL 2 syntax employs rules for the resolution of relative IRIs (e.g., the OWL 2 XML Syntax [OWL 2 XML Syntax] uses *xml:base* for that purpose), such rules do not apply to *xsd:anyURI* lexical values that represent relative IRIs; that is, the lexical values representing relative IRIs *must* be parsed as they are.

Facet Space. The facet space of the *xsd:anyURI* datatype is shown in Table 7.

Table 7. The Facet Space of the *xsd:anyURI* Datatype

Pair	Facet Value
〈 *xsd:minLength v* 〉 where *v* is a nonnegative integer	the set of IRIs *I* from the value space of *xsd:anyURI* such that the length of the string representation of *I* is at least *v*

〈 *xsd:maxLength v* 〉 where *v* is a nonnegative integer	the set of IRIs *I* from the value space of *xsd:anyURI* such that the length of the string representation of *I* is at most *v*
〈 *xsd:length v* 〉 where *v* is a nonnegative integer	the set of IRIs *I* from the value space of *xsd:anyURI* such that the length of the string representation of *I* is exactly *v*
〈 *xsd:pattern v* 〉 where *v* is a string regular expression with the syntax as in Section F of XML Schema Datatypes [XML Schema Datatypes]	the set of IRIs *I* from the value space of *xsd:anyURI* whose string representation matches the regular expression *v*

4.6 Time Instants

OWL 2 provides the *owl:dateTime* datatype for the representation of time instants. This datatype is equivalent to the *xsd:dateTime* datatype of XML Schema Datatypes [XML Schema Datatypes] with a required timezone.

Feature At Risk #3: *owl:dateTime* name

The name *owl:dateTime* is currently a placeholder. XML Schema 1.1 Working Group will introduce a datatype for date-time with required timezone. Once this is done, *owl:dateTime* will be changed to whatever name XML Schema chooses. If the schedule of the XML Schema 1.1 Working Group slips the OWL 2 Working Group will consider possible alternatives.

Please send feedback to public-owl-comments@w3.org.

Value Space. The value space of *owl:dateTime* is the set of numbers, where each number *x* represents the time instant occurring *x* seconds after the first time instant of the 1st of January 1 AD in the proleptic Gregorian calendar [*ISO 8601:2004*] (i.e., the calendar in which the Gregorian dates are retroactively applied to the dates preceding the introduction of the Gregorian calendar). This set can be seen as a "copy" of the set of real numbers — that is, it is disjoint with but isomorphic to the value space of *owl:real*. For simplicity, the elements from this set can be identified with real numbers.

Lexical Space. The *owl:dateTime* datatype supports lexical values defined by the following grammar (whitespace within the grammar *must* be ignored and *must not* be included in the lexical values of *owl:dateTime*, and single quotes are used to introduce terminal symbols):

```
year '-' month '-' date 'T' hour ':' minute ':' second
timezone
```

The components of the this string are as follows:

- Characters `-` (U+2D), `T` (U+54), and `:` (U+3A) separate the various parts of the string.
- `year` is an integer consisting of at least four decimal digits optionally preceded by a minus sign; leading zero digits are prohibited except to bring the digit count up to four.
- `month`, `day`, `hour`, and `minute` are integers consisting of exactly two decimal digits.
- `second` is an integer consisting of exactly two decimal digits, or two decimal digits, a decimal point, and one or more trailing digits.
- `timezone` specifies a count of minutes that has to be added to or subtracted from UTC in order to get local time. The grammar for `timezone` is given by the following three alternatives, each of which is mapped to an integer as specified next:
    - a time point of the form `'+' tzHours ':' tzMinutes` between `+00:00` (inclusive) and `+14:00` (exclusive) is mapped to the value `+ tzHours × 60 + tzMinutes`;
    - a time point of the form `'-' tzHours ':' tzMinutes` between `-00:00` (inclusive) and `-14:00` (exclusive) is mapped to the value `- tzHours × 60 + tzMinutes`;
    - `'Z'` is mapped to the value `0`.
- `month` is between 1 and 12 (inclusive).
- `day` is no more than 31 if `month` is one of 1, 3, 5, 7, 8, 10, or 12; no more than 30 if `month` is one of 4, 6, 9, or 11; no more than 29 if `month` is 2 and `year` is divisible by 400, or by 4 but not by 100; and no more than 28 if `month` is 2 and `year` is not divisible 4, or is divisible by 100 but not by 400.
- `hour`, `minute`, and `second` represent a time point between `00:00:00` (inclusive) and `24:00:00` (exclusive).

Each such lexical value is assigned a data value as specified by the following function, where `div` represents integer division and `mod` is the remainder of integer division. This mapping does not take into account leap seconds: leap seconds will be introduced in UTC as deemed necessary in future; since the precise date when this will be done is not known, the OWL 2 specification ignores leaps seconds.

dataValue(year, month, day, hour, minutes, seconds, timezone) =	
31536000 × (year-1) +	# convert all previous years to seconds
86400 × ((year-1) div 400 - (year-1) div 100 + (year-1) div 4) +	# adjust for leap years
86400 × $\text{Sum}_{m < \text{month}}$ daysInMonth(year, m) +	# add the duration of each month
86400 × (day-1) +	# add the duration of the previous days

3600 × hour + 60 × (minutes - timezone) + seconds	# add the current time

daysInMonth(y, m) =

28	if m = 2 and [(y mod 4 ≠ 0) or (y mod 100 = 0 and y mod 400 ≠ 0)]
29	if m = 2 and [(y mod 400 = 0) or (y mod 4 = 0 and y mod 100 ≠ 0)]
30	if m ∈ { 4, 6, 9, 11 }
31	if m ∈ { 1, 3, 5, 7, 8, 10, 12 }

Lexical values of *owl:dateTime* can represent an arbitrary date. An OWL 2 implementation *may* support all such lexical values; however, it *must* support at least all lexical values in which the absolute value of the `year` component is less than 10000 (i.e., whose representation requires at most four digits), and in which the `second` component is a number with at most three decimal digits.

Facet Space. The facet space of the *owl:dateTime* datatype is shown in Table 8.

Table 8. The Facet Space of the *owl:dateTime* Datatype

Pair	Facet Value
〈 *xsd:minInclusive v* 〉 where *v* is from the value space of *owl:dateTime*	the set of all time instants *x* from the value space of *owl:dateTime* such that *x* = *v* or *x* > *v*
〈 *xsd:maxInclusive v* 〉 where *v* is from the value space of *owl:dateTime*	the set of all time instants *x* from the value space of *owl:dateTime* such that *x* = *v* or *x* < *v*
〈 *xsd:minExclusive v* 〉 where *v* is from the value space of *owl:dateTime*	the set of all time instants *x* from the value space of *owl:dateTime* such that *x* > *v*
〈 *xsd:maxExclusive v* 〉 where *v* is from the value space of *owl:dateTime*	the set of all time instants *x* from the value space of *owl:dateTime* such that *x* < *v*

4.7 XML Literals

OWL 2 uses the *rdf:XMLLiteral* datatype for the representation of XML content in OWL 2 ontologies. The definitions of the value space, the lexical space, and the mapping from the lexical to the value space are given in Section 5.1 of the RDF specification [RDF]. The *rdf:XMLLiteral* datatype supports no constraining facets.

Feature At Risk #4: *rdf:XMLLiteral* support

Note: This feature is "at risk" and may be removed from this specification based on feedback. Please send feedback to public-owl-comments@w3.org.

The *rdf:XMLLiteral* datatype might be removed from OWL 2 if implementation experience reveals problems with supporting this datatype.

5 Entities and Literals

Entities are the fundamental building blocks of OWL 2 ontologies, and they define the vocabulary — the named terms — of an ontology. In logic, the set of entities is usually said to constitute the *signature* of an ontology. Apart from entities, OWL 2 ontologies typically also contain literals, such as strings or integers.

The structure of entities and literals in OWL 2 is shown in Figure 2. Classes, datatypes, object properties, data properties, annotation properties, and named individuals are entities, and they are all uniquely identified by an IRI. Classes can be used to model sets of individuals; datatypes are sets of literals such as strings or integers; object and data properties can be used to represent relationships in the modeled domain; annotation properties can be used to associate nonlogical information with ontologies, axioms, and entities; and named individuals can be used to represent actual objects from the domain being modeled. Apart from named individuals, OWL 2 also provides for anonymous individuals — that is, individuals that are analogous to blank nodes in RDF [RDF Syntax] and that are accessible only from within the ontology they are used in. Finally, OWL 2 provides for literals, which consist of a lexical value and a datatype specifying how to interpret this value.

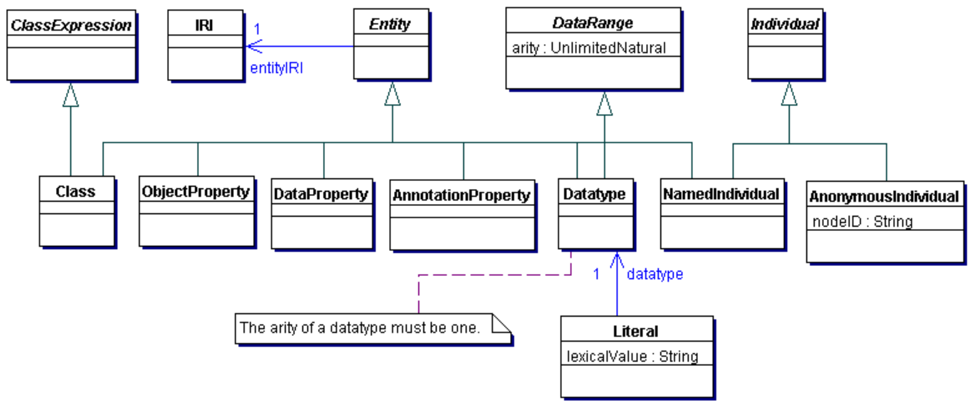

Figure 2. The Hierarchy of Entities in OWL 2

5.1 Classes

Classes can be understood as sets of individuals.

Class := **IRI**

IRIs used to identify classes *must not* be in the reserved vocabulary, apart from *owl:Thing* and *owl:Nothing*, which are available in OWL 2 as built-in classes with a predefined semantics.

- The class with IRI *owl:Thing* represents the set of all individuals. (In the DL literature this is often called the top concept.)
- The class with IRI *owl:Nothing* represents the empty set. (In the DL literature this is often called the bottom concept.)

Example:

Classes *a:Child* and *a:Person* can be used to model the set of all children and persons, respectively, in the application domain, and they can be used in an axiom such as the following one:

`SubClassOf( a:Child a:Person )`	Each child is a person.

5.2 Datatypes

Datatypes are entities that refer to sets of values described by a datatype map (see Section 4). Thus, datatypes are analogous to classes, the main difference being that the former contain values such as strings and numbers, rather than individuals. Datatypes are a kind of data ranges, which allows them to be used in restrictions. All datatypes have arity one. The built-in datatype *rdfs:Literal* denotes any set that contains the union of the value spaces of all datatypes in the datatype map. Each datatype other than *rdfs:Literal must* belong to the datatype map.

Datatype := **IRI**

Example:

The datatype *xsd:integer* denotes the set of all integers. It can be used in axioms such as the following one:

`PropertyRange( a:hasAge xsd:integer )`	The range of the *a:hasAge* property is *xsd:integer*.

5.3 Object Properties

Object properties connect pairs of individuals.

ObjectProperty := **IRI**

IRIs used to identify object properties *must not* be in the reserved vocabulary, apart from *owl:topObjectProperty* and *owl:bottomObjectProperty*, which are available in OWL 2 as built-in object properties with a predefined semantics.

- The object property with IRI *owl:topObjectProperty* connects all possible pairs of individuals. (In the DL literature this is often called the top role.)
- The object property with IRI *owl:bottomObjectProperty* does not connect any pair of individuals. (In the DL literature this is often called the bottom role.)

Example:

The object property *a:parentOf* can be used to represent the parenthood relationship between individuals. It can be used in axioms such as the following one:

`PropertyAssertion( a:parentOf a:Peter a:Chris )`	Peter is a parent of Chris.

5.4 Data Properties

Data properties connect individuals with literals. In some knowledge representation systems, functional data properties are called *attributes*.

DataProperty := **IRI**

IRIs used to identify data properties *must not* be in the reserved vocabulary, apart from *owl:topDataProperty* and *owl:bottomDataProperty*, which are are available in OWL 2 as built-in data properties with a predefined semantics.

- The data property with IRI *owl:topDataProperty* connects all possible individuals with all literals. (In the DL literature this is often called the top role.)
- The data property with IRI *owl:bottomDataProperty* does not connect any individual with a literal. (In the DL literature this is often called the bottom role.)

Example:

The data property *a:hasName* can be used to associate a name with each person. It can be used in axioms such as the following one:

`PropertyAssertion( a:hasName a:Peter "Peter Griffin" )`	Peter's name is `"Peter Griffin"`.

5.5 Annotation Properties

Annotation properties can be used to provide an annotation for an ontology, axiom, or an IRI. The structure of annotations is further described in Section 10.

AnnotationProperty := **IRI**

IRIs used to identify annotation properties *must not* be in the reserved vocabulary, apart from the following IRIs from the reserved vocabulary, which are are available in OWL 2 as built-in annotation properties.

- The *rdfs:label* annotation property can be used to provide an IRI with a human-readable label.
- The *rdfs:comment* annotation property can be used to provide an IRI with a human-readable comment.
- The *rdfs:seeAlso* annotation property can be used to provide an IRI with another IRI such that the latter provides additional information about the former.
- The *rdfs:isDefinedBy* annotation property can be used to provide an IRI with another IRI such that the latter provides information about the definition of the former; the way in which this information is provided is not described by this specification.
- An annotation with the *owl:deprecated* annotation property and the value equal to `"true"^^xsd:boolean` can be used to specify that an IRI is deprecated.
- The *owl:priorVersion* annotation property is described in more detail in Section 3.5.
- The *owl:backwardCompatibleWith* annotation property is described in more detail in Section 3.5.
- The *owl:incompatibleWith* annotation property is described in more detail in Section 3.5.

Example:

The comment provided by the following annotation assertion axiom might, for example, be used by an OWL 2 tool to display additional information about the IRI *a:Peter*.

`AnnotationAssertion( rdfs:comment a:Peter "The father of the Griffin family from Quahog." )`	This axiom provides a comment for the IRI *a:Peter*.

5.6 Individuals

Individuals represent actual objects from the domain being modeled. There are two types of individuals in OWL 2. *Named individuals* are given an explicit name that can be used in any ontology in the import closure to refer to the same individual. *Anonymous individuals* are local to the ontology they are contained in.

Individual := **NamedIndividual** | **AnonymousIndividual**

5.6.1 Named Individuals

Named individuals are identified using an IRI. Since they are given an IRI, named individuals are entities. IRIs used to identify named individuals *must not* be in the reserved vocabulary.

NamedIndividual := **IRI**

Example:

The individual *a:Peter* can be used to represent a particular person. It can be used in axioms such as the following one:

`ClassAssertion( a:Person a:Peter )`	Peter is a person.

5.6.2 Anonymous Individuals

If an individual is not expected to be used outside an ontology, one can model it as an *anonymous individual*, which is identified by a local node ID. Anonymous individuals are analogous to blank nodes in RDF [RDF Syntax].

AnonymousIndividual := **nodeID**

Example:

Anonymous individuals can be used, for example, to represent objects whose identity is of no relevance, such as the address of a person.

`PropertyAssertion( a:livesAt a:Peter _:1 )`	Peter lives at some (unknown) address.
`PropertyAssertion( a:city _:1 a:Quahog )`	This unknown address is in the city of Quahog and...
`PropertyAssertion( a:state _:1 a:RI )`	...in the state of Rhode Island.

Special treatment is required in case anonymous individuals with the same node ID occur in two different ontologies. In particular, these two individuals are structurally equivalent (because they have the same node ID); however, they are treated as different individuals in the semantics of OWL 2 (because anonymous individuals are local to an ontology they are used in). The latter is achieved by *renaming anonymous individuals apart* when constructing the axiom closure of an ontology *O*: if anonymous individuals with the same node ID occur in two different ontologies in the import closure of *O*, then one of these individuals *must* be replaced in the axiom closure of *O* with a fresh anonymous individual (i.e., with an anonymous individual having a globally unique node ID).

Example:

Assume that ontologies O_1 and O_2 both use _:*a5*, and that O_1 imports O_2. Although they both use the same local node ID, the individual _:*a5* in O_1 may be different from the individual _:*a5* in O_2.

At the level of the structural specification, individual _:*a5* in O_1 is structurally equivalent to individual _:*a5* in O_2. This might be important, for example, for tools that use structural equivalence to define the semantics of axiom retraction.

In order to ensure that these individuals are treated differently by the semantics they are renamed apart when computing the axiom closure of O_1 — either _:*a5* in O_1 is replaced with a fresh anonymous individual, or this is done for _:*a5* in O_2.

5.7 Literals

Literals represent values such as particular strings or integers. They are analogous to literals in RDF [RDF Syntax] and can also be understood as individuals denoting known data values. Each literal consists of a lexical value, which is a string, and a datatype. The lexical value *must* conform to restrictions as specified by the datatype in the datatype map. The datatype map also determines how the literal is mapped to the actual data value. The datatypes and literals supported in OWL 2 are described in more detail in Section 4.

Literals are generally written in the functional-style syntax as `"abc"^^datatype`. The functional-style also supports the abbreviations for common types of text literals [RDF:TEXT], and OWL 2 implementations *should* use these abbreviated forms whenever possible. These abbreviations are purely syntactic shortcuts and are thus not reflected in the structural specification of OWL 2.

- Literals of the form `"abc"^^`*`xsd:string`* *should* be abbreviated to `"abc"`.
- Literals of the form `"abc@languageTag"^^`*`rdf:text`* *should* be abbreviated to `"abc"@`*`languageTag`*.

Literal := **typedLiteral** | **abbreviatedXSDStringLiteral** | **abbreviatedRDFTextLiteral**
typedLiteral := **lexicalValue** '^^' **Datatype**
lexicalValue := **quotedString**
abbreviatedXSDStringLiteral := **quotedString**
abbreviatedRDFTextLiteral := **quotedString** '@' **languageTag**

Example:

`"1"^^`*`xsd:integer`* is a literal that represents the integer 1.

Example:

`"Family Guy"` is an abbreviation for `"Family Guy"^^`*`xsd:string`* — a literal with the lexical value `"Family Guy"` and the datatype *xsd:string*.

Example:

`"Padre de familia"@es` is an abbreviation for the literal `"Padre de familia@es"^^`*`rdf:text`* — a literal denoting a pair consisting of the string `"Padre de familia"` and the language tag `es` denoting the Spanish language.

Two literals are structurally equivalent if and only if both the lexical value and the datatype are structurally equivalent; that is, literals denoting the same data value are structurally different if either their lexical value or the datatype is different.

Example:

Even through literals `"1"^^`*`xsd:integer`* and `"+1"^^`*`xsd:integer`* are interpreted as the integer 1, these two literals are not structurally equivalent because their lexical values are not the same. Similarly, `"1"^^`*`xsd:integer`* and `"1"^^xsd:positiveInteger` are not structurally equivalent because their datatypes are not the same.

5.8 Entity Declarations and Typing

Each IRI *I* used in an OWL 2 ontology *O* can, and sometimes even must, be declared in *O*; roughly speaking, this means that the axiom closure of *O* must contain an appropriate declaration for *I*. A declaration for *I* in *O* serves two purposes:

- A declaration says that *I* exists — that is, it says that *I* is part of the vocabulary of *O*.
- A declaration associates with *I* an entity type — that is, it says whether *I* is used in *O* as a class, datatype, object property, data property, annotation property, an individual, or a combination thereof.

Example:

An ontology might contain a declaration for the IRI *a:Person* and state that this IRI is a class. Such a declaration states that *a:Person* exists in the ontology and it states that the IRI is used as a class. An ontology editor might use declarations to implement functions such as "Add New Class".

In OWL 2, declarations are a type of axiom; thus, to declare an entity in an ontology, one can simply include the appropriate axiom in the ontology. These axioms are nonlogical in the sense that they do not affect the direct semantics of an OWL 2 ontology [*OWL 2 Direct Semantics*]. The structure of entity declarations is shown in Figure 3.

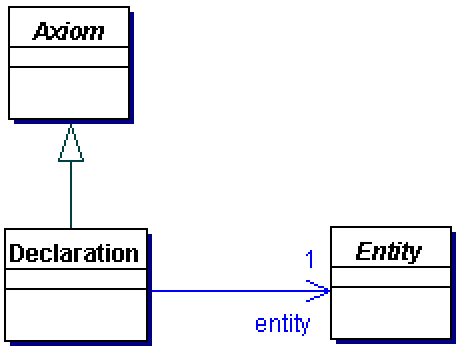

Figure 3. Entity Declarations in OWL 2

```
Declaration := 'Declaration' '(' axiomAnnotations Entity ')'
Entity :=
    'Class' '(' Class ')' |
    'Datatype' '(' Datatype ')' |
    'ObjectProperty' '(' ObjectProperty ')' |
    'DataProperty' '(' DataProperty ')' |
    'AnnotationProperty' '(' AnnotationProperty ')' |
    'NamedIndividual' '(' NamedIndividual ')'
```

Example:

The following axioms state that the IRI *a:Person* is used as a class and that the IRI *a:Peter* is used as an individual.

```
Declaration( Class( a:Person ) )
Declaration( NamedIndividual( a:Peter ) )
```

Declarations for the built-in entities of OWL 2, listed in Table 9, are implicitly present in every OWL 2 ontology.

Table 9. Declarations of Built-In Entities

Declaration(Class(*owl:Thing*))
Declaration(Class(*owl:Nothing*))
Declaration(ObjectProperty(*owl:topObjectProperty*))
Declaration(ObjectProperty(*owl:bottomObjectProperty*))
Declaration(DataProperty(*owl:topDataProperty*))
Declaration(DataProperty(*owl:bottomDataProperty*))
Declaration(Datatype(*rdfs:Literal*))

`Declaration( Datatype( I ) )`	for each IRI *I* of a datatype in the datatype map (see Section 4)
`Declaration( AnnotationProperty( I ) )`	for each IRI *I* of a built-in annotation property listed in Section 5.5

5.8.1 Typing Constraints

Let *Ax* be a set of axioms. An IRI *I* is *declared* to be of type *T* in *Ax* if a declaration axiom of type *T* for *I* is contained in *Ax* or in the set of built-in declarations listed in Table 9. The set *Ax* satisfies the *typing constraints* of OWL 2 if all of the following conditions are satisfied:

- Property typing constraints:
    - If an object property with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as an object property.
    - If a data property with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as a data property.
    - If an annotation property with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as an annotation property.
    - No IRI *I* is declared in *Ax* as being of more than one type of property; that is, no *I* is declared in *Ax* to be both object and data, object and annotation, or data and annotation property.
- Class/datatype typing constraints:
    - If a class with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as a class.
    - If a datatype with an IRI *I* occurs in some axiom in *Ax*, then *I* is declared in *Ax* as a datatype.
    - No IRI *I* is declared in *ax* to be both a class and a datatype.
- No declaration for an IRI *I* violates the constraints on the usage of reserved vocabulary listed in the previous sections.

The axiom closure *Ax* of each OWL 2 ontology *O must* satisfy the typing constraints of OWL 2.

The typing constraints thus ensure that the sets of IRIs used as object, data, and annotation properties in *O* are disjoint and that, similarly, the sets of IRIs used as classes and datatypes in *O* are disjoint as well. These constraints are used for disambiguating the types of IRIs when reading ontologies from external transfer syntaxes. All other declarations are optional.

Example:

An IRI *I* can be used as an individual in *O* even if *I* is not declared as an individual in *O*.

Declarations are often omitted in the examples in this document in cases where the types of entities are clear.

5.8.2 Declaration Consistency

Although declarations are optional for the most part, they can be used to catch obvious errors in ontologies.

Example:

The following ontology erroneously refers to the individual *a:Petre* instead of the individual *a:Peter*.

```
Ontology(<http://www.my.domain.com/example>
    ClassAssertion( a:Person a:Petre )
)
```

There is no way of telling whether *a:Petre* was used by mistake. If, in contrast, all individuals in an ontology were by convention required to be declared, this error could be caught by a simple tool.

An ontology *O* is said to have *consistent declarations* if each IRI *I* occurring in the axiom closure of *O* in position of an entity with a type *T* is declared in *O* as having type *T*. OWL 2 ontologies are not required to have consistent declarations: an ontology *may* be used even if its declarations are not consistent.

Example:

The ontology from the previous example fails this check: *a:Petre* is used as an individual but the ontology does not declare *a:Petre* to be an individual, and similarly for *a:Person*. In contrast, the following ontology satisfies this condition.

```
Ontology(<http://www.my.domain.com/example>
    Declaration( Class( a:Person ) )
    Declaration( NamedIndividual( a:Peter ) )
    ClassAssertion( a:Person a:Peter )
)
```

5.9 Metamodeling

According to the typing constraints from Section 5.8.1, an IRI *I* can be used in an OWL 2 ontology to refer to more than one type of entity. Such usage of *I* is often called *metamodeling*, because it can be used to state facts about classes and properties themselves. In such cases, the entities that share the same IRI *I* should be understood as different "views" of the same underlying notion identified by the IRI *I*.

Example:

Consider the following ontology.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`ClassAssertion( a:Species a:Dog )`	Dog is a species.

In the first axiom, the IRI *a:Dog* is used as a class, while in the second axiom, it is used as an individual; thus, the class *a:Species* acts as a metaclass for the class *a:Dog*. The individual *a:Dog* and the class *a:Dog* should be understood as two "views" of one and the same IRI — *a:Dog*. Under the OWL 2 Direct Semantics [*OWL 2 Direct Semantics*], these two views are interpreted independently: the class view of *a:Species* is interpreted as a unary predicate, while the individual view of *a:Species* is interpreted as a constant.

Both metamodeling and annotations provide means to associate additional information with classes and properties. The following rule-of-the-thumb can be used to determine when to use which construct:

- Metamodeling should be used when the information attached to entities should be considered a part of the domain being modeled.
- Annotations should be used when the information attached to entities should not be considered a part of the domain being modeled and when it should not contribute to the logical consequences of an ontology.

Example:

Consider the following ontology.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`ClassAssertion( a:PetAnimals a:Dog )`	Dogs are pet animals.
`AnnotationAssertion( a:addedBy a:Dog "Seth MacFarlane" )`	The IRI *a:Dog* has been added to the ontology by Seth MacFarlane.

The facts that Brian is a dog and that dogs are pet animals are statements about the domain being modeled. Therefore, these facts are represented in the above ontology via metamodeling. In contrast, the information about who added the IRI *a:Dog* to the ontology does not describe the actual domain being modeled, but might be interesting from a management point of view. Therefore, this information is represented using an annotation.

6 Property Expressions

Properties can be used in OWL 2 to form property expressions.

6.1 Object Property Expressions

Object properties can by used in OWL 2 to form object property expressions. They are represented in the structural specification of OWL 2 by **ObjectPropertyExpression**, and their structure is shown in Figure 4.

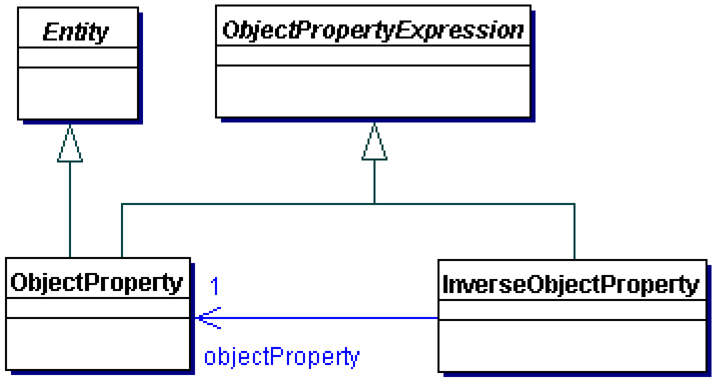

Figure 4. Object Property Expressions in OWL 2

As one can see from the figure, OWL 2 supports only two kinds of object property expressions. Object properties are the simplest form of object property expressions, and inverse object properties allow for bidirectional navigation in class expressions and axioms.

```
ObjectPropertyExpression := ObjectProperty | InverseObjectProperty
```

6.1.1 Inverse Object Properties

An inverse object property expression `InverseOf( P )` connects an individual I_1 with I_2 if and only if the object property `P` connects I_2 with I_1.

```
InverseObjectProperty := 'InverseOf' '(' ObjectProperty ')'
```

Example:

Consider the ontology consisting of the following assertion.

`PropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is the father of Stewie.

This ontology entails that *a:Stewie* is connected via `InverseOf( a:fatherOf )` to *a:Peter*.

6.2 Data Property Expressions

For symmetry with object property expressions, the structural specification of OWL 2 also introduces the notion of data property expressions, as shown in Figure 5. The only allowed data property expression is a data property; thus, **DataPropertyExpression** in the structural specification of OWL 2 can be seen as a place-holder for possible future extensions.

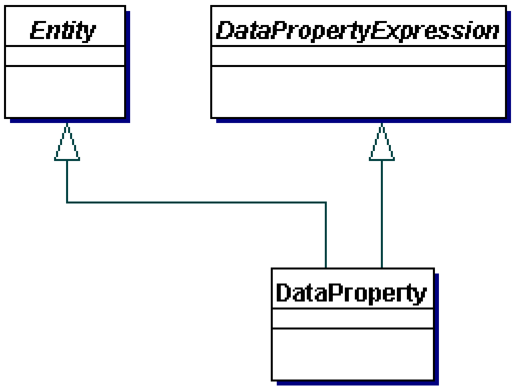

Figure 5. Data Property Expressions in OWL 2

DataPropertyExpression := **DataProperty**

7 Data Ranges

Datatypes, such as strings or integers, can be used to express data ranges — sets of tuples of literals. Each data range is associated with a positive arity, which determines the size of the tuples in the data range. All datatypes have arity one. This specification currently does not define data ranges of arity more than one; however, by allowing for *n*-ary data ranges, the syntax of OWL 2 provides a "hook" allowing implementations to introduce extensions such as comparisons and arithmetic.

Data ranges can be used in restrictions on data properties, as discussed in Sections 8.4 and 8.5. The structure of data ranges in OWL 2 is shown in Figure 6. The simplest data ranges are datatypes. The **DataIntersectionOf**, **DataUnionOf**, and **DataComplementOf** data ranges provide for the standard set-theoretic operations on data ranges; in logical languages these are usually called conjunction, disjunction, and negation, respectively. The **DataOneOf** data range consists of exactly the specified set of literals. Finally, the **DatatypeRestriction** data range restricts the value space of a datatype by a constraining facet.

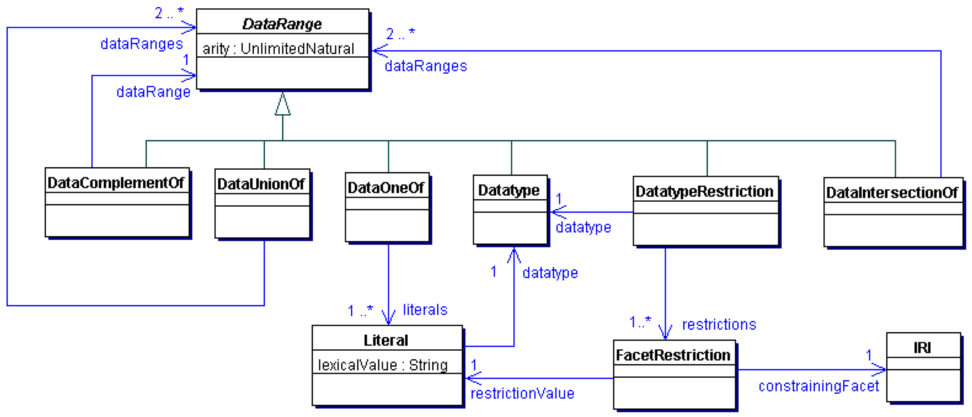

Figure 6. Data Ranges in OWL 2

DataRange :=
  Datatype |
  DataIntersectionOf |
  DataUnionOf |
  DataComplementOf |
  DataOneOf |
  DatatypeRestriction

7.1 Intersection of Data Ranges

An intersection data range `IntersectionOf(` DR_1 `...` DR_n `)` contains all data values that are contained in the value space of every data range DR_i for 1 ≤ i ≤ n. All data ranges DR_i must be of the same arity.

DataIntersectionOf := `'IntersectionOf'` `'('` **DataRange** **DataRange** { **DataRange** } `')'`

Example:

The data range `IntersectionOf( xsd:nonNegativeInteger xsd:nonPositiveInteger )` contains exactly the integer 0.

7.2 Union of Data Ranges

A union data range `UnionOf(` DR_1 `...` DR_n `)` contains all data values that are contained in the value space of at least one data range DR_i for 1 ≤ i ≤ n. All data ranges DR_i must be of the same arity.

DataUnionOf := 'UnionOf' '(' **DataRange** **DataRange** { **DataRange** } ')'

Example:

The data range `UnionOf( xsd:string xsd:integer )` contains all strings and all integers.

7.3 Complement of Data Ranges

A complement data range `ComplementOf( DR )` contains all literals that are not contained in the data range `DR`.

DataComplementOf := 'ComplementOf' '(' **DataRange** ')'

Example:

The complement data range `ComplementOf( xsd:positiveInteger )` consists of literals that are not positive integers. In particular, this data range contains the integer zero and all negative integers; however, it also contains all strings (since strings are not positive integers).

7.4 Enumeration of Literals

An enumeration of literals `OneOf(` lt_1 `...` lt_n `)` contains exactly the explicitly specified literals lt_i with 1 ≤ i ≤ n.

DataOneOf := 'OneOf' '(' **Literal** { **Literal** } ')'

Example:

The enumeration of literals `OneOf( "Peter" "1"^^xsd:integer )` contains exactly two literals: the string `"Peter"` and the integer one.

7.5 Datatype Restrictions

A datatype restriction `DatatypeRestriction(` $DT\ F_1\ lt_1\ \ldots\ F_n\ lt_n$ `)` consists of a unary datatype `DT` and `n` pairs $\langle F_i\ lt_i \rangle$. Let v_i be the data values of the corresponding literals lt_i. Each pair $\langle F_i\ v_i \rangle$ *must* be contained in the facet space of `DT` in the datatype map (see Section 4). The resulting unary data range is obtained by restricting the value space of `DT` according to the semantics of all $\langle F_i\ v_i \rangle$ (multiple pairs are interpreted conjunctively).

DatatypeRestriction := 'DatatypeRestriction' '(' **Datatype** **constrainingFacet** **restrictionValue** { **constrainingFacet** **restrictionValue** } ')'
constrainingFacet := **IRI**
restrictionValue := **Literal**

Example:

The data range `DatatypeRestriction( xsd:integer xsd:minInclusive "5"^^xsd:integer xsd:maxExclusive "10"^^xsd:integer )` contains exactly the integers 5, 6, 7, 8, and 9.

8 Class Expressions

In OWL 2, classes and property expressions are used to construct *class expressions*, sometimes also called *descriptions*, and, in the description logic literature, *complex concepts*. Class expressions represent sets of individuals by formally specifying conditions [*OWL 2 Direct Semantics*] on the individuals' properties; individuals satsifying these conditions are said to be *instances* of the respective class expressions. In the structural specification of OWL 2, class expressions are represented by **ClassExpression**.

Example:

A class expression can be used to represent the set of "people that have at least one child". If an ontology additionally contains statements that "Peter is a person" and that "Peter has child Chris", then Peter can be classified as an instance of the mentioned class expression.

OWL 2 provides a rich set of primitives that can be used to construct class expressions. In particular, it provides the well known Boolean connectives *and*, *or*, and *not*; a restricted form of universal and existential quantification; number restrictions; enumeration of individuals; and a special *self*-restriction.

As shown in Figure 2, classes are the simplest form of class expressions. The other, complex, class expressions, are described in the following sections.

```
ClassExpression  :=
    Class  |
    ObjectIntersectionOf  |  ObjectUnionOf  |  ObjectComplementOf  |
ObjectOneOf  |
    ObjectSomeValuesFrom  |  ObjectAllValuesFrom  |  ObjectHasValue  |
ObjectHasSelf  |
    ObjectMinCardinality  |  ObjectMaxCardinality  |  ObjectExactCardinality
|
    DataSomeValuesFrom  |  DataAllValuesFrom  |  DataHasValue  |
    DataMinCardinality  |  DataMaxCardinality  |  DataExactCardinality
```

8.1 Propositional Connectives and Enumeration of Individuals

OWL 2 provides for enumeration of individuals and all standard Boolean connectives, as shown in Figure 7. The **ObjectIntersectionOf**, **ObjectUnionOf**, and **ObjectComplementOf** class expressions provide for the standard set-theoretic operations on class expressions; in logical languages these are usually called conjunction, disjunction, and negation, respectively. The **ObjectOneOf** class expression contains exactly the specified individuals.

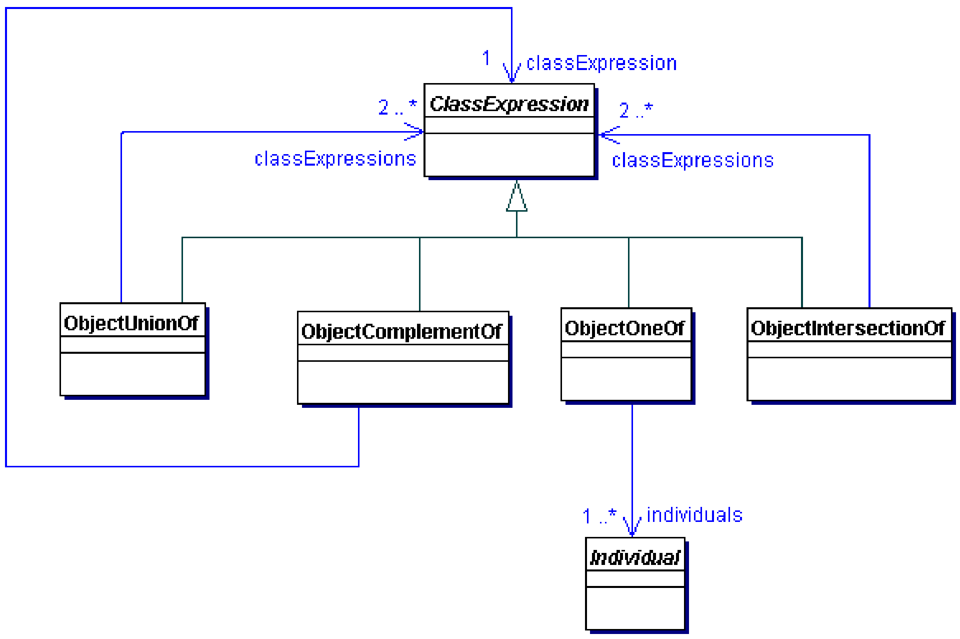

Figure 7. Propositional Connectives and Enumeration of Individuals in OWL 2

8.1.1 Intersection of Class Expressions

An intersection class expression `IntersectionOf( CE`$_1$ `... CE`$_n$ `)` contains all individuals that are instances of all class expressions `CE`$_i$ for 1 ≤ i ≤ n.

ObjectIntersectionOf := `'IntersectionOf'` `'('` **ClassExpression** **ClassExpression** `{` **ClassExpression** `}` `')'`

Example:

Consider the ontology consisting of the following axioms.

`ClassAssertion(` *`a:Dog a:Brian`* `)`	Brian is a dog.
`ClassAssertion(` *`a:CanTalk a:Brian`* `)`	Brian can talk.

The class expression `IntersectionOf(` *`a:Dog a:CanTalk`* `)` describes all dogs that can talk and, consequently, *a:Brian* is classified as an instance of this expression.

8.1.2 Union of Class Expressions

A union class expression `UnionOf( CE1 ... CEn )` contains all individuals that are instances of at least one class expression CE_i for 1 ≤ i ≤ n.

ObjectUnionOf := `'UnionOf'` `'('` **ClassExpression** **ClassExpression** `{` **ClassExpression** `}` `')'`

Example:

Consider the ontology consisting of the following axioms.

`ClassAssertion( a:Man a:Peter )`	Peter is a man.
`ClassAssertion( a:Woman a:Lois )`	Lois is a woman.

The class expression `UnionOf( a:Man a:Woman )` describes all individuals that are instances of either *a:Man* or *a:Woman*; consequently, both *a:Peter* and *a:Lois* are classified as instances of this expression.

8.1.3 Complement of Class Expressions

A complement class expression `ComplementOf( CE )` contains all individuals that are not instances of the class expression `CE`.

ObjectComplementOf := `'ComplementOf'` `'('` **ClassExpression** `')'`

Example:

Consider the ontology consisting of the following axioms.

`DisjointClasses( a:Man a:Woman )`	Nothing can be both a man and a woman.
`ClassAssertion( a:Woman a:Lois )`	Lois is a woman.

The class expression `ComplementOf( a:Man )` describes all things that are not instances of *a:Man*. Since *a:Lois* is known to be a woman and nothing can be both a man and a woman, then *a:Lois* is necessarily not a *a:Man*; therefore, *a:Lois* is classified as an instance of this complement class expression.

Example:

OWL 2 has *open-world* semantics, so negation in OWL 2 is the same as in classical (first-order) logic. To understand open-world semantics, consider the ontology consisting of the following assertion.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.

One might expect *a:Brian* to be classified as an instance of `ComplementOf( a:Bird )`: the ontology does not explicitly state that *a:Brian* is an instance of *a:Bird*, so this statement seems to be false. In OWL 2, however, this is not the case: it is true that the ontology does not state that *a:Brian* is an instance of *a:Bird*; however, the ontology does not state the opposite either. In other words, this ontology simply does not contain enough information to answer the question whether *a:Brian* is an instance of *a:Bird* or not: it is perfectly possible that the information to that effect is actually true but it has not been included in the ontology.

The ontology from the previous example (in which *a:Lois* has been classified as *a:Man*), however, contains sufficient information to draw the expected conclusion. In particular, we know for sure that *a:Lois* is an instance of *a:Woman* and that *a:Man* and *a:Woman* do not share instances. Therefore, any additional information that does not lead to inconsistency cannot lead to a conclusion that *a:Lois* is an instance of *a:Man*; furthermore, if one were to explicitly state that *a:Lois* is an instance of *a:Man*, the ontology would be inconsistent and, by definition, it then entails all possible conclusions.

8.1.4 Enumeration of Individuals

An enumeration of individuals `OneOf(` a_1 `...` a_n `)` contains exactly the individuals a_i with 1 ≤ i ≤ n.

ObjectOneOf := `'OneOf'` `'('` **Individual** `{` **Individual** `}')'`

Example:

Consider the ontology consisting of the following axioms.

`EquivalentClasses( a:GriffinFamilyMember OneOf( a:Peter a:Lois a:Stewie a:Meg a:Chris a:Brian ) )`	The Griffin family consists exactly of Peter, Lois, Stewie, Meg, and Brian.

`DifferentIndividuals( a:Quagmire a:Peter a:Lois a:Stewie a:Meg a:Chris a:Brian )`	Quagmire, Peter, Lois, Stewie, Meg, Chris, and Brian are all different from each other.

The class *a:GriffinFamilyMember* now contains exactly the six explicitly listed individuals. Since we also know that *a:Quagmire* is different from these six individuals, this individual is classified as an instance of the class expression `ComplementOf( a:GriffinFamilyMember )`. The last axiom is necessary to derive this conclusion; without it, the open-world semantics of OWL 2 would allow for situations where *a:Quagmire* is the same as *a:Peter*, *a:Lois*, *a:Stewie*, *a:Meg*, *a:Chris*, or *a:Brian*.

Example:

To understand how the open-world semantics affects enumerations of individuals, consider the ontology consisting of the following axioms.

`ClassAssertion( a:GriffinFamilyMember a:Peter )`	Peter is a member of the Griffin Family.
`ClassAssertion( a:GriffinFamilyMember a:Lois )`	Lois is a member of the Griffin Family.
`ClassAssertion( a:GriffinFamilyMember a:Stewie )`	Stewie is a member of the Griffin Family.
`ClassAssertion( a:GriffinFamilyMember a:Meg )`	Meg is a member of the Griffin Family.
`ClassAssertion( a:GriffinFamilyMember a:Chris )`	Chris is a member of the Griffin Family.
`ClassAssertion( a:GriffinFamilyMember a:Brian )`	Brian is a member of the Griffin Family.

The class *a:GriffinFamilyMember* now also contains the mentioned six individuals, just as in the previous example. The main difference to the previous example, however, is that the extension of *a:GriffinFamilyMember* is not closed: the semantics of OWL 2 assumes that information about a potential instance of *a:GriffinFamilyMember* may be missing. Therefore, *a:Quagmire* is now not classified as an instance of the class expression `ComplementOf( a:GriffinFamilyMember )`, and this does not change even if we add the axiom stating that all of these six individuals are different from each other.

8.2 Object Property Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on object property expressions, as shown in Figure 8. The **ObjectSomeValuesFrom** class expression allows for existential quantification over an object property expression, and it contains those individuals that are connected through an object property expression to at least one instance of a given class expression. The **ObjectAllValuesFrom** class expression allows for universal quantification over an object property expression, and it contains those individuals that are connected through an object property expression only to instances of a given class expression. The **ObjectHasValue** class expression contains those individuals that are connected by an object property expression to a particular individual. Finally, the **ObjectHasSelf** class expression contains those individuals that are connected by an object property expression to themselves.

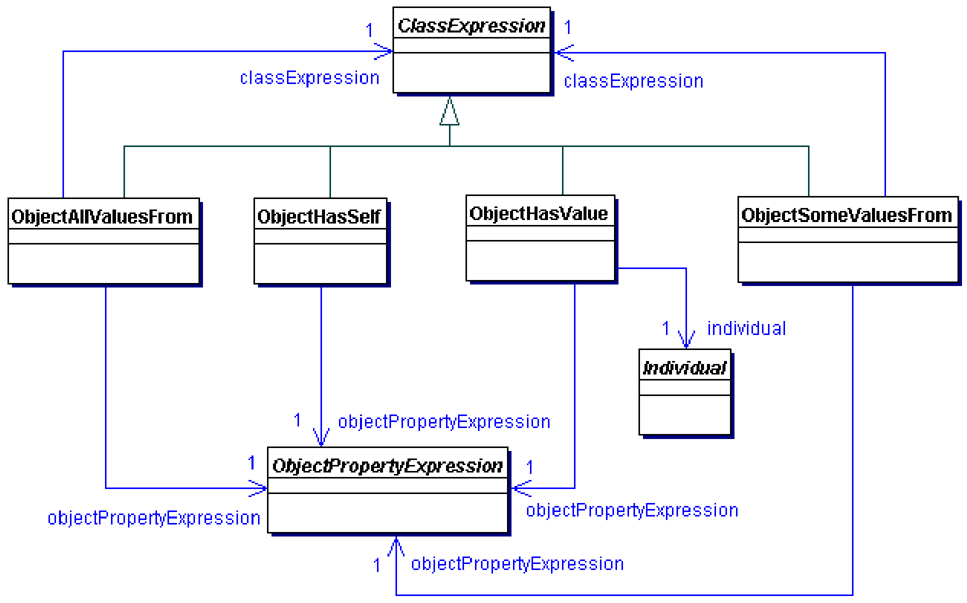

Figure 8. Restricting Object Property Expressions in OWL 2

8.2.1 Existential Quantification

An existential class expression `SomeValuesFrom( OPE CE )` consists of an object property expression `OPE` and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to an individual that is an instance of `CE`. Provided that `OPE` is *simple* according to the definition in Section 11, such a class expression can be seen as a syntactic shortcut for the class expression `MinCardinality( 1 OPE CE )`.

ObjectSomeValuesFrom := 'SomeValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is the father of Stewie.
`ClassAssertion( a:Man a:Stewie )`	Stewie is a man.

The existential expression `SomeValuesFrom( a:fatherOf a:Man )` contains those individuals that are connected by the *a:fatherOf* property to individuals that are instances of *a:Man* and, consequently, *a:Peter* is classified as an instance of this class expression.

8.2.2 Universal Quantification

A universal class expression `AllValuesFrom( OPE CE )` consists of an object property expression `OPE` and a class expression `CE`, and it contains all those individuals that are connected by `OPE` only to individuals that are instances of `CE`. Provided that `OPE` is *simple* according to the definition in Section 11, such a class expression can be seen as a syntactic shortcut for the class expression `MaxCardinality( 0 OPE ComplementOf( CE ) )`.

ObjectAllValuesFrom := 'AllValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:hasPet a:Peter a:Brian )`	Brian is a pet of Peter.
`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`ClassAssertion( MaxCardinality( 1 a:hasPet ) a:Peter )`	Peter has at most one pet.

The universal expression `AllValuesFrom( a:hasPet a:Dog )` contains those individuals that are connected through the *a:hasPet* property only with individuals that are instances of *a:Dog*; in other words, it contains individuals that have only dogs as pets. The ontology axioms clearly state that *a:Peter* is connected by *a:hasPet* only to instances of *a:Dog*: it is impossible to connect *a:Peter* by *a:hasPet* to an individual different from *a:Brian* without making the ontology inconsistent. Therefore, *a:Peter* is classified as an instance of `AllValuesFrom( a:hasPet a:Dog )`.

The last axiom — that is, the axiom stating that *a:Peter* has at most one pet — is critical for the inference from the previous paragraph due to the open-world semantics of OWL 2. Without this axiom, the ontology might not have listed all the individuals to which *a:Peter* is connected by *a:hasPet*. In such a case *a:Peter* would not be classified as an instance of `AllValuesFrom( a:hasPet a:Dog )`.

8.2.3 Individual Value Restriction

A has-value class expression `HasValue( OPE a )` consists of an object property expression `OPE` and an individual `a`, and it contains all those individuals that are connected by `OPE` to `a`. Each such class expression can be seen as a syntactic shortcut for the class expression `SomeValuesFrom( OPE OneOf( a ) )`.

ObjectHasValue `:=` `'HasValue'` `'('` **ObjectPropertyExpression** **Individual** `')'`

Example:

Consider the ontology consisting of the following axiom.

`PropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is the father of Stewie.

The has-value class expression `HasValue( a:fatherOf a:Stewie )` contains those individuals that are connected through the *a:fatherOf* property with the individual *a:Stewie* so, consequently, *a:Peter* is classified as an instance of this class expression.

8.2.4 Self-Restriction

A self-restriction `HasSelf( OPE )` consists of an object property expression `OPE`, and it contains all those individuals that are connected by `OPE` to themselves.

ObjectHasSelf `:=` `'HasSelf'` `'('` **ObjectPropertyExpression** `')'`

Example:

Consider the ontology consisting of the following axiom.

`PropertyAssertion( a:likes a:Peter a:Peter )`	Peter likes himself.

The self-restriction `HasSelf( a:likes )` contains those individuals that like themselves so, consequently, *a:Peter* is classified as an instance of this class expression.

8.3 Object Property Cardinality Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on the cardinality of object property expressions, as shown in Figure 9. All cardinality restrictions can be qualified or unqualified: in the former case, the cardinality restriction only applies to individuals that are connected by the object property expression and are instances of the qualifying class expression; in the latter case the restriction applies to all individuals that are connected by the object property expression (this is equivalent to the qualified case with the qualifying class expression equal to *owl:Thing*). The class expressions **ObjectMinCardinality**, **ObjectMaxCardinality**, and **ObjectExactCardinality** contain those individuals that are connected by an object property expression to at least, at most, and exactly a given number of instances of a specified class expression, respectively.

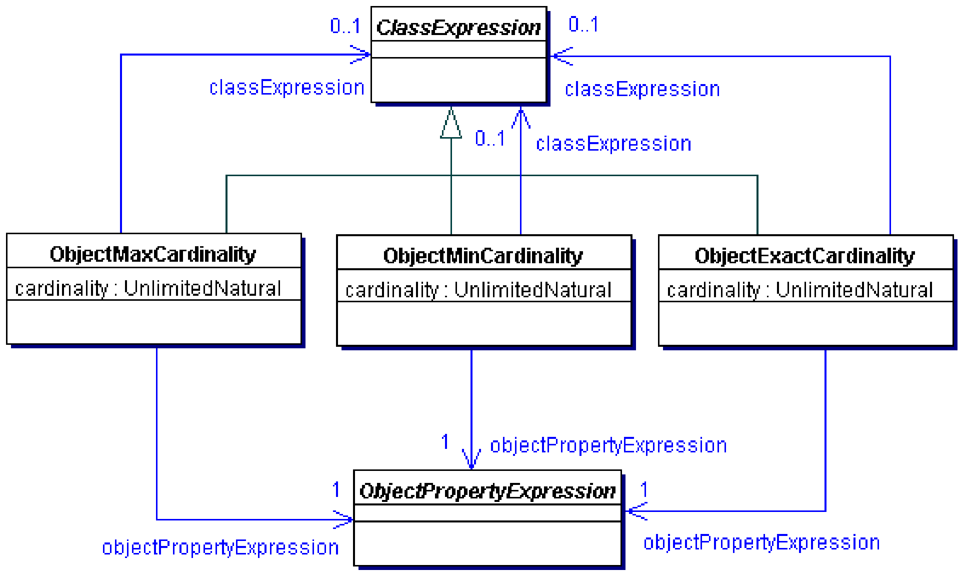

Figure 9. Restricting the Cardinality of Object Property Expressions in OWL 2

8.3.1 Minimum Cardinality

A minimum cardinality expression `MinCardinality( n OPE CE )` consists of a nonnegative integer `n`, an object property expression `OPE`, and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to at least `n` different individuals that are instances of `CE`. If `CE` is missing, it is taken to be *owl:Thing*.

ObjectMinCardinality `:=` `'MinCardinality'` `'('` **nonNegativeInteger** **ObjectPropertyExpression** `[` **ClassExpression** `]` `')'`

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is the father of Stewie.
`ClassAssertion( a:Man a:Stewie )`	Stewie is a man.
`PropertyAssertion( a:fatherOf a:Peter a:Chris )`	Peter is the father of Chris.
`ClassAssertion( a:Man a:Chris )`	Chris is a man.
`DifferentIndividuals( a:Chris a:Stewie )`	Chris and Stewie are different from each other.

The minimum cardinality expression `MinCardinality( 2 a:fatherOf a:Man )` contains those individuals that are connected by *a:fatherOf* to at least two different instances of *a:Man*. Since *a:Stewie* and *a:Chris* are both instances of *a:Man* and are different from each other, *a:Peter* is classified as an instance of `MinCardinality( 2 a:fatherOf a:Man )`.

Due to the open-world semantics, the last axiom — stating that *a:Chris* and *a:Stewie* are different from each other — is necessary for this inference: without this axiom, it is possible that *a:Chris* and *a:Stewie* are actually the same individual.

8.3.2 Maximum Cardinality

A maximum cardinality expression `MaxCardinality( n OPE CE )` consists of a nonnegative integer `n`, an object property expression `OPE`, and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to at most `n` different individuals that are instances of `CE`. If `CE` is missing, it is taken to be *owl:Thing*.

ObjectMaxCardinality := 'MaxCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:hasPet a:Peter a:Brian )`	Brian is a pet of Peter.
`ClassAssertion( MaxCardinality( 1 a:hasPet ) a:Peter )`	Peter has at most one pet.

The maximum cardinality expression `MaxCardinality( 2 a:hasPet )` contains those individuals that are connected by *a:hasPet* to at most two individuals. Since *a:Peter* is known to be connected by *a:hasPet* to at most one individual, it is certainly also connected by *a:hasPet* to at most two individuals so, consequently, *a:Peter* is classified as an instance of `MaxCardinality( 2 a:hasPet )`.

The example ontology explicitly names only *a:Brian* as being connected by *a:hasPet* from *a:Peter*, so one might expect *a:Peter* to be classified as an instance of `MaxCardinality( 2 a:hasPet )` even without the second axiom. This, however, is not the case due to the open-world semantics. Without the last axiom, it is possible that *a:Peter* is connected by *a:hasPet* to other individuals. The second axiom closes the set of individuals that *a:Peter* is connected to by *a:hasPet*.

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:hasDaughter a:Peter a:Meg )`	Meg is a daughter of Peter.
`PropertyAssertion( a:hasDaughter a:Peter a:Megan )`	Megan is a daughter of Peter.
`ClassAssertion( MaxCardinality( 1 a:hasDaughter ) a:Peter )`	Peter has at most one daughter.

One might expect this ontology to be inconsistent: on the one hand, it says that *a:Meg* and *a:Megan* are connected to *a:Peter* by *a:hasDaughter*, but, on the other hand, it says that *a:Peter* is connected by *a:hasDaughter* to at most one individual. This ontology, however, is not inconsistent because the semantics of OWL 2 does not make the *unique name assumption* — that is, it does not assume distinct individuals to be necessarily different. For example, the ontology does not explicitly say that *a:Meg* and *a:Megan* are different individuals; therefore, since *a:Peter* can be connected by *a:hasDaughter* to at most one distinct individual, *a:Meg* and *a:Megan* must be the same. This example ontology thus entails the assertion `SameIndividual( a:Meg a:Megan )`.

One can axiomatize the unique name assumption in OWL 2 by explicitly stating that all individuals are different from each other. This can be done by adding the following axiom, which makes the example ontology inconsistent.

`DifferentIndividuals( a:Peter a:Meg a:Megan )`	Peter, Meg, and Megan are all different from each other.

8.3.3 Exact Cardinality

An exact cardinality expression `ExactCardinality( n OPE CE )` consists of a nonnegative integer `n`, an object property expression `OPE`, and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to exactly `n` different individuals that are instances of `CE`. If `CE` is missing, it is taken to be *owl:Thing*. Such an expression is actually equivalent to the expression

```
IntersectionOf( MinCardinality( n OPE CE ) MaxCardinality(
n OPE CE ) ).
```

ObjectExactCardinality := 'ExactCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

Example:

Consider the ontology consisting of the following axioms.

```
PropertyAssertion( a:hasPet
a:Peter a:Brian )                      Brian is a pet of Peter.
ClassAssertion( a:Dog a:Brian )        Brian is a dog.
ClassAssertion(
    AllValuesFrom( a:hasPet
       UnionOf(
          OneOf( a:Brian )
          ComplementOf( a:Dog )        Each pet of Peter is either
       )                               Brian or it is not a dog.
    )
    a:Peter
)
```

The exact cardinality expression `ExactCardinality( 1 a:hasPet a:Dog )` contains those individuals that are connected by *a:hasPet* to exactly one instance of *a:Dog*. The example ontology says that *a:Peter* is connected to *a:Brian* by *a:hasPet* and that *a:Brian* is an instance of *a:Dog*; therefore, *a:Peter* is an instance of `MinCardinality( 1 a:hasPet a:Dog )`. Furthermore, the last axiom says that any individual different from *a:Brian* that is connected to *a:Peter* by *a:hasPet* is not an instance if *a:Dog*; therefore, *a:Peter* is an instance of `MaxCardinality( 1 a:hasPet a:Dog )`. Consequently, *a:Peter* is classified as an instance of `ExactCardinality( 1 a:hasPet a:Dog )`.

8.4 Data Property Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on data property expressions, as shown in Figure 10. These are similar to the restrictions on object property expressions, the main difference being that the expressions for existential and universal quantification allow for *n*-ary data ranges. All data ranges explicitly supported by this specification are unary; however, the provision of *n*-ary data ranges in existential and universal quantification allows OWL 2 tools to support extensions such as value comparisons and, consequently, class expressions such as "individuals whose width is greater than their height". Thus, the **DataSomeValuesFrom** class expression allows for a restricted existential quantification over a list of data property expressions, and it contains those individuals that are connected through the data property expressions to at least one literal in the given data range. The **DataAllValuesFrom** class expression allows for a restricted universal quantification over a list of data property expressions, and it contains those individuals that are connected through the data property expressions only to literals in the given data range. Finally, the **DataHasValue** class expression contains those individuals that are connected by a data property expression to a particular literal.

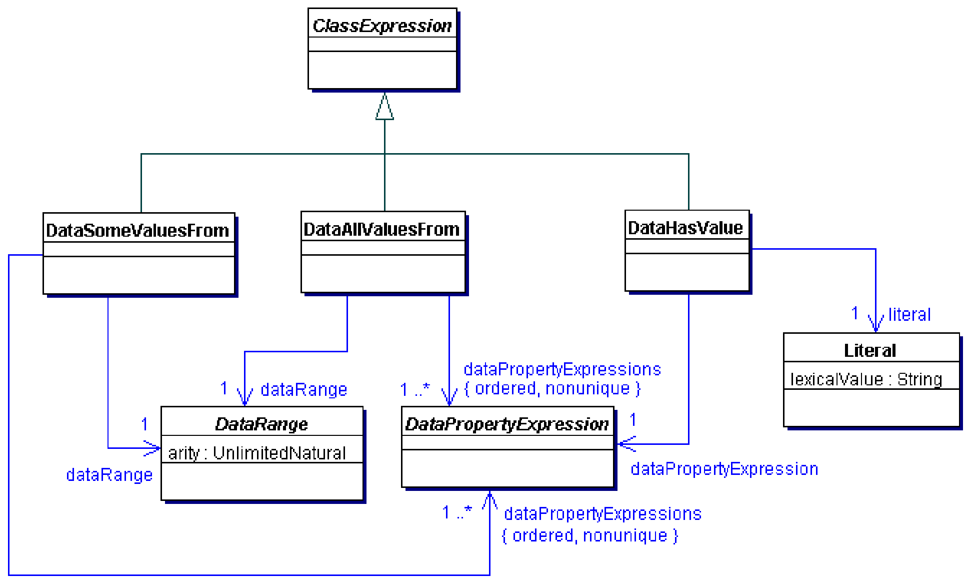

Figure 10. Restricting Data Property Expressions in OWL 2

8.4.1 Existential Quantification

An existential class expression `SomeValuesFrom( DPE`$_1$` ... DPE`$_n$` DR )` consists of `n` data property expressions `DPE`$_i$, $1 \leq i \leq n$, and a data range `DR` whose arity *must* be n. Such a class expression contains all those individuals that are connected by `DPE`$_i$ to literals `lt`$_i$, $1 \leq i \leq n$, such that the tuple 〈 `lt`$_1$`, ..., lt`$_n$ 〉 is in `DR`. A class expression of the form `SomeValuesFrom( DPE DR )` can be seen as a syntactic shortcut for the class expression `MinCardinality( 1 DPE DR )`.

DataSomeValuesFrom := 'SomeValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

Example:

Consider the ontology consisting of the following axiom.

`PropertyAssertion( a:hasAge a:Meg "17"^^xsd:integer )`	Meg is seventeen years old.

The existential class expression `SomeValuesFrom( a:hasAge DatatypeRestriction( xsd:integer xsd:maxExclusive "20"^^xsd:integer ) )` contains all individuals that are connected by *a:hasAge* to an integer strictly less than 20 so, consequently, *a:Meg* is classified as an instance of this expression.

8.4.2 Universal Quantification

A universal class expression `AllValuesFrom( DPE`$_1$` ... DPE`$_n$` DR )` consists of `n` data property expressions `DPE`$_i$, 1 ≤ i ≤ n, and a data range `DR` whose arity *must* be n. Such a class expression contains all those individuals that are connected by `DPE`$_i$ only to literals `lt`$_i$, 1 ≤ i ≤ n, such that each tuple ⟨ `lt`$_1$`, ..., lt`$_n$ ⟩ is in `DR`. A class expression of the form `AllValuesFrom( DPE DR )` can be seen as a syntactic shortcut for the class expression `MaxCardinality( 0 DPE ComplementOf( DR ) )`.

DataAllValuesFrom := 'AllValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:zipCode _:a1 "02903"^^xsd:integer )`	The ZIP code of _:a1 is the integer 02903.
`FunctionalProperty( a:hasZIP )`	Each object can have at most one ZIP code.

In United Kingdom and Canada, ZIP codes are strings (i.e., they can contain characters and not just numbers). Hence, one might use the universal expression `AllValuesFrom( a:hasZIP xsd:integer )` to identify those individuals that have only integer ZIP codes (and therefore have non-UK and non-Canadian addresses). The anonymous individual _:a1 is by the first axiom connected by *a:zipCode* to an integer, and the second axiom ensures that _:a1 is not connected by *a:zipCode* to other literals; therefore, _:a1 is classified as an instance of `AllValuesFrom( a:hasZIP xsd:integer )`.

The last axiom — stating that *a:hasZIP* is functional — is critical for the inference from the previous paragraph due to the open-world semantics of OWL 2. Without this axiom, the ontology is not guaranteed to list all literals that _:a1 is connected to by *a:hasZIP*; hence, without this axiom _:a1 would not be classified as an instance of `AllValuesFrom( a:hasZIP xsd:integer )`.

8.4.3 Literal Value Restriction

A has-value class expression `HasValue( DPE lt )` consists of a data property expression `DPE` and a literal `lt`, and it contains all those individuals that are connected by `DPE` to `lt`. Each such class expression can be seen as a syntactic shortcut for the class expression `SomeValuesFrom( DPE OneOf( lt ) )`.

DataHasValue `:=` `'HasValue'` `'('` **DataPropertyExpression** **Literal** `')'`

Example:

Consider the ontology consisting of the following axiom.

`PropertyAssertion(` *`a:hasAge`* *`a:Meg`* `"17"^^`*`xsd:integer`* `)`	Meg is seventeen years old.

The has-value expression `hasValue(` *`a:hasAge`* `"17"^^`*`xsd:integer`* `)` contains all individuals that are connected by *a:hasAge* to the integer 17 so, consequently, *a:Meg* is classified as an instance of this expression.

8.5 Data Property Cardinality Restrictions

Class expressions in OWL 2 can be formed by placing restrictions on the cardinality of data property expressions, as shown in Figure 11. These are similar to the restrictions on the cardinality of object property expressions. All cardinality restrictions can be qualified or unqualified: in the former case, the cardinality restriction only applies to literals that are connected by the data property expression and are in the qualifying data range; in the latter case it applies to all literals that are connected by the data property expression (this is equivalent to the qualified case with the qualifying data range equal to *rdfs:Literal*). The class expressions **DataMinCardinality**, **DataMaxCardinality**, and **DataExactCardinality** contain those individuals that are connected by a data property expression to at least, at most, and exactly a given number of literals in the specified data range, respectively.

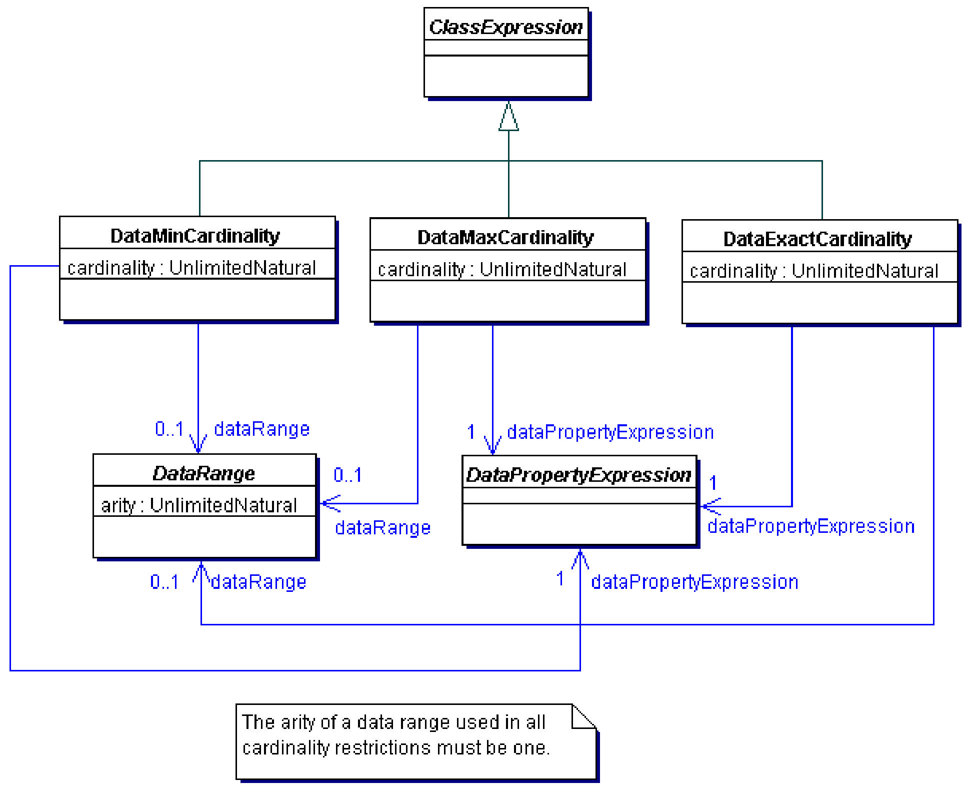

Figure 11. Restricting the Cardinality of Data Property Expressions in OWL 2

8.5.1 Minimum Cardinality

A minimum cardinality expression `MinCardinality( n DPE DR )` consists of a nonnegative integer `n`, a data property expression `DPE`, and a unary data range `DR`, and it contains all those individuals that are connected by `DPE` to at least `n` different literals in `DR`. If `DR` is not present, it is taken to be *rdfs:Literal*.

DataMinCardinality := 'MinCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:hasName a:Meg "Meg Griffin" )`	Meg's name is "`Meg Griffin`".

`PropertyAssertion( a:hasName a:Meg "Megan Griffin" )`	Meg's name is `"Megan Griffin"`.

The minimum cardinality expression `MinCardinality( 2 a:hasName )` contains those individuals that are connected by *a:hasName* to at least two different literals. The *xsd:string* datatypes interprets different string literals as being distinct, so `"Meg Griffin"` and `"Megan Griffin"` are different; thus, the individual *a:Meg* is classified as an instance of the class expression `MinCardinality( 2 a:hasName )`.

8.5.2 Maximum Cardinality

A maximum cardinality expression `MaxCardinality( n DPE DR )` consists of a nonnegative integer `n`, a data property expression `DPE`, and a unary data range `DR`, and it contains all those individuals that are connected by `DPE` to at most `n` different literals in `DR`. If `DR` is not present, it is taken to be *rdfs:Literal*.

DataMaxCardinality `:= 'MaxCardinality' '('` **nonNegativeInteger DataPropertyExpression** `[` **DataRange** `] ')'`

Example:

Consider the ontology consisting of the following axiom.

`FunctionalProperty( a:hasName )`	Each object can have at most one name.

The maximum cardinality expression `MaxCardinality( 2 a:hasName )` contains those individuals that are connected by *a:hasName* to at most two different literals. Since the ontology axiom restricts *a:hasName* to be functional, all individuals in the ontology are instances of this class expression.

8.5.3 Exact Cardinality

An exact cardinality expression `ExactCardinality( n DPE DR )` consists of a nonnegative integer `n`, a data property expression `DPE`, and a unary data range `DR`, and it contains all those individuals that are connected by `DPE` to exactly `n` different literals in `DR`. If `DR` is not present, it is taken to be *rdfs:Literal*.

DataExactCardinality := 'ExactCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:hasName a:Brian "Brian Griffin" )`	Brian's name is `"Brian Griffin"`.
`FunctionalProperty( a:hasName )`	Each object can have at most one name.

The exact cardinality expression `ExactCardinality( 1 a:hasName )` contains those individuals that are connected by *a:hasName* to exactly one literal. Since the ontology axiom restricts *a:hasName* to be functional and *a:Brian* is connected by *a:hasName* to `"Brian Griffin"`, it is classified as an instance of this class expression.

9 Axioms

The main component of an OWL 2 ontology is a set of *axioms* — statements that say what is true in the domain being modeled. OWL 2 provides an extensive set of axioms, all of which extend the **Axiom** class in the structural specification. As shown in Figure 12, axioms in OWL 2 can be declarations, axioms about classes, axioms about object or data properties, keys, assertions (sometimes also called *facts*), and axioms about annotations.

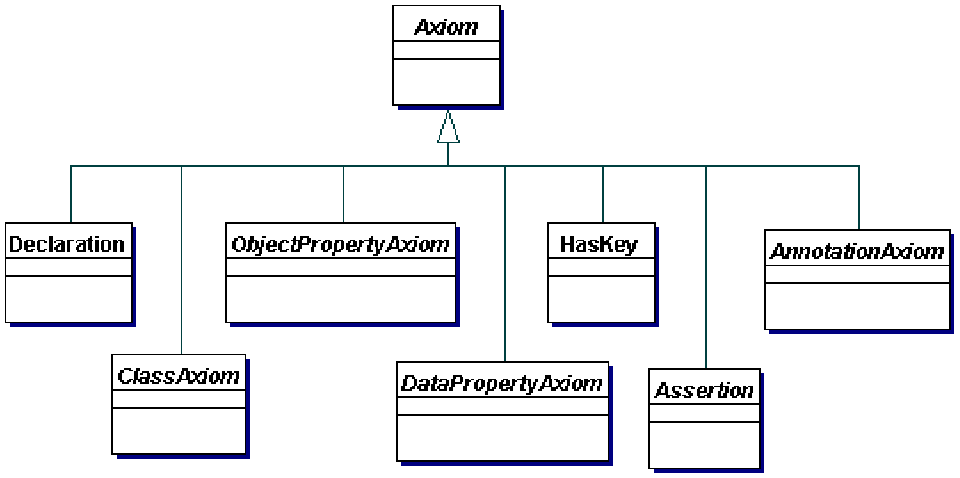

Figure 12. The Axioms of OWL 2

Axiom := **Declaration** | **ClassAxiom** | **ObjectPropertyAxiom** | **DataPropertyAxiom** | **HasKey** | **Assertion** | **AnnotationAxiom**

axiomAnnotations := { **Annotation** }

As shown in Figure 1, OWL 2 axioms can contain axiom annotations, the structure of which is defined in Section 10. Axiom annotations have no effect on the semantics of axioms — that is, they do not affect the meaning of OWL 2 ontologies [*OWL 2 Direct Semantics*]. In contrast, axiom annotations do affect structural equivalence: axioms will not be structurally equivalent if their axiom annotations are not structurally equivalent.

Example:

The following axiom contains a comment that explains the purpose of the axiom.

```
SubClassOf( Annotation( rdfs:comment "Male people are
people.") a:Man a:Person)
```

Since annotations affect structural equivalence between axioms, the previous axiom is not structurally equivalent with the following axiom, even though these two axioms are equivalent according to the OWL 2 Direct Semantics [*OWL 2 Direct Semantics*].

```
SubClassOf( a:Man a:Person )
```

9.1 Class Expression Axioms

OWL 2 provides axioms that allow relationships to be established between class expressions, as shown in Figure 13. The **SubClassOf** axiom allows one to state that each instance of one class expression is also an instance of another class expression, and thus to construct a hierarchy of classes. The **EquivalentClasses** axiom allows one to state that several class expressions are equivalent to each other. The **DisjointClasses** axiom allows one to state that several class expressions are pairwise disjoint — that is, that they have no instances in common. Finally, the **DisjointUnion** class expression allows one to define a class as a disjoint union of several class expressions and thus to express covering constraints.

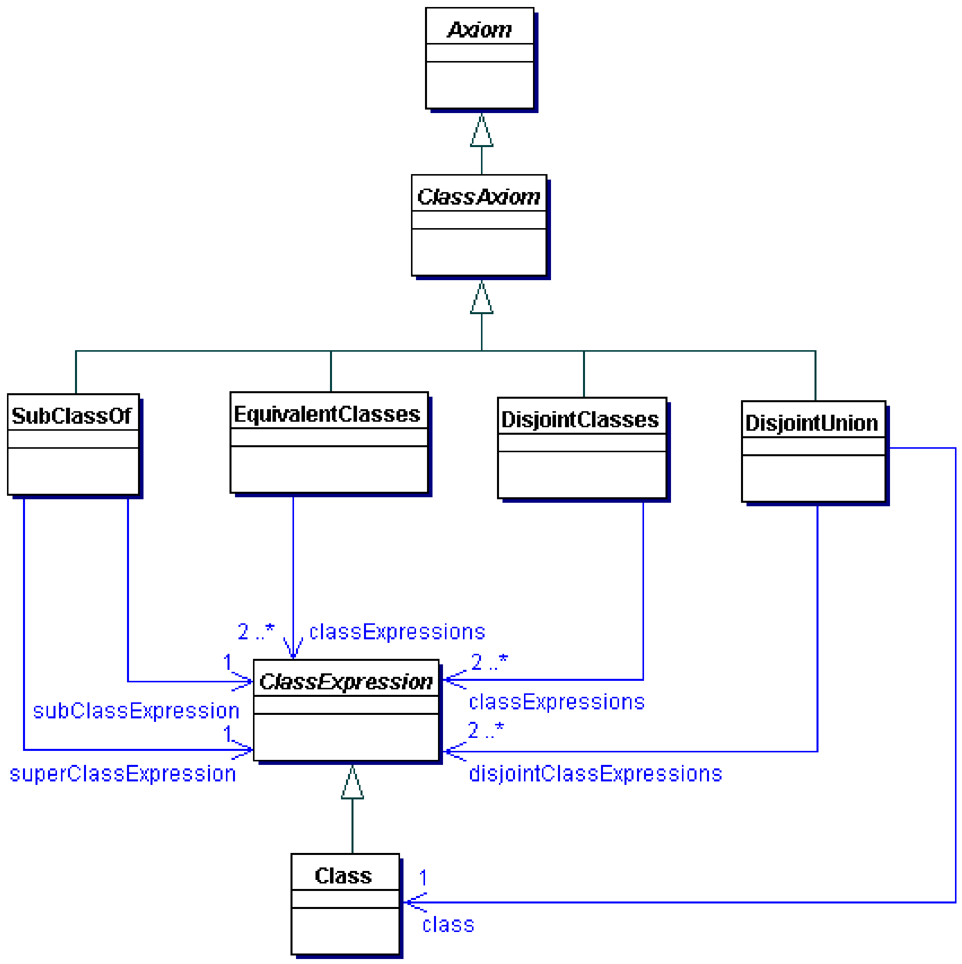

Figure 13. The Class Axioms of OWL 2

ClassAxiom := **SubClassOf** | **EquivalentClasses** | **DisjointClasses** | **DisjointUnion**

9.1.1 Subclass Axioms

A subclass axiom `SubClassOf( CE`$_1$ `CE`$_2$ `)` states that the class expression `CE`$_1$ is a subclass of the class expression `CE`$_2$. Roughly speaking, this states that `CE`$_1$ is more specific than `CE`$_2$. Subclass axioms are a fundamental type of axioms in OWL 2 and can be used to construct a class hierarchy. Other kinds of class expression axiom can be seen as syntactic shortcuts for one or more subclass axioms.

SubClassOf `:= 'SubClassOf' '('` **axiomAnnotations subClassExpression superClassExpression** `')'`
subClassExpression `:=` **ClassExpression**
superClassExpression `:=` **ClassExpression**

Example:

Consider the ontology consisting of the following axioms.

`SubClassOf( a:Baby a:Child )`	Each baby is a child.
`SubClassOf( a:Child a:Person )`	Each child is a person.
`ClassAssertion( a:Baby a:Stewie )`	Stewie is a baby.

Since *a:Stewie* is an instance of *a:Baby*, by the first subclass axiom *a:Stewie* is classified as an instance of *a:Child* as well. Similarly, by the second subclass axiom *a:Stewie* is classified as an instance of *a:Person*. This style of reasoning can be applied to any instance of *a:Baby* and not just *a:Stewie*; therefore, one can conclude that *a:Baby* is a subclass of *a:Person*. In other words, this ontology entails the axiom `SubClassOf( a:Baby a:Person )`.

Example:

Consider the ontology consisting of the following axioms.

`SubClassOf( a:PersonWithChild SomeValuesFrom( a:hasChild UnionOf( a:Boy a:Girl ) ) )`	A person that has a child has either at least one boy or a girl.
`SubClassOf( a:Boy a:Child )`	Each boy is a child.
`SubClassOf( a:Girl a:Child )`	Each girl is a child.
`SubClassOf( SomeValuesFrom( a:hasChild a:Child ) a:Parent )`	If some object has a child, then this object is a parent.

The first axiom states that each instance of *a:PersonWithChild* is connected to an individual that is an instance of either *a:Boy* or *a:Girl*. (Because of the open-world semantics of OWL 2, this does not mean that there must be only one such individual or that all such individuals must be instances of either *a:Boy* or of *a:Girl*.) Furthermore, each instance of *a:Boy* or *a:Girl* is an instance of *a:Child*. Finally, the last axiom says that all individuals that are connected by *a:hasChild* to an instance of *a:Child* are instances of *a:Parent*. Since this reasoning holds for each instance of *a:PersonWithChild*, each such instance is also an instance of *a:Parent*. In other words, this ontology entails the axiom `SubClassOf( a:PersonWithChild a:Parent )`.

9.1.2 Equivalent Classes

An equivalent classes axiom EquivalentClasses(CE_1 ... CE_n) states that all of the class expressions CE_i, 1 ≤ i ≤ n, are semantically equivalent to each other. This axiom allows one to use each CE_i as a synonym for each CE_j — that is, in any expression in the ontology containing such an axiom, CE_i can be replaced with CE_j without affecting the meaning of the ontology. An axiom EquivalentClasses(CE_1 CE_2) is equivalent to the following two axioms:

SubClassOf(CE_1 CE_2)
SubClassOf(CE_2 CE_1)

Axioms of the form `EquivalentClasses( C CE )`, where `C` is a class and `CE` is a class expression, are often called *definitions*, because they define the class `C` in terms of the class expression `CE`.

EquivalentClasses := 'EquivalentClasses' '(' **axiomAnnotations** **ClassExpression** **ClassExpression** { **ClassExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

`EquivalentClasses( a:Boy IntersectionOf( a:Child a:Man ) )`	A boy is a male child.
`ClassAssertion( a:Child a:Chris )`	Chris is a child.
`ClassAssertion( a:Man a:Chris )`	Chris is a man.
`ClassAssertion( a:Boy a:Stewie )`	Stewie is a boy.

The first axiom defines the class *a:Boy* as an intersection of the classes *a:Child* and *a:Man*; thus, the instances of *a:Boy* are exactly those instances that are both an instance of *a:Child* and an instance of *a:Man*. Such a definition consists of two directions. The first direction implies that each instance of *a:Child* and *a:Man* is an instance of *a:Boy*; since *a:Chris* satisfies these two conditions, it is classified as an instance of *a:Boy*. The second direction implies that each *a:Boy* is an instance of *a:Child* and of *a:Man*; thus, *a:Stewie* is classified as an instance of *a:Man* and of *a:Boy*.

Example:

Consider the ontology consisting of the following axioms.

`EquivalentClasses( a:MongrelOwner SomeValuesFrom( a:hasPet a:Mongrel ) )`	A mongrel owner has a pet that is a mongrel.
`EquivalentClasses( a:DogOwner SomeValuesFrom( a:hasPet a:Dog ) )`	A dog owner has a pet that is a dog.
`SubClassOf( a:Mongrel a:Dog )`	Each mongrel is a dog.
`ClassAssertion( a:MongrelOwner a:Peter )`	Peter is a mongrel owner.

By the first axiom, each instance `x` of *a:MongrelOwner* must be connected via *a:hasPet* to an instance of *a:Mongrel*; by the third axiom, this individual is an instance of *a:Dog*; thus, by the second axiom, `x` is an instance of *a:DogOwner*. In other words, this ontology entails the axiom `SubClassOf( a:MongrelOwner a:DogOwner )`. By the fourth axiom, *a:Peter* is then classified as an instance of *a:DogOwner*.

9.1.3 Disjoint Classes

A disjoint classes axiom DisjointClasses(CE_1 ... CE_n) states that all of the class expressions CE_i, 1 ≤ i ≤ n, are pairwise disjoint; that is, no individual can be at the same time an instance of both CE_i and CE_j for i ≠ j. An axiom DisjointClasses(CE_1 CE_2) is equivalent to the following axiom:

SubClassOf(CE_1 ComplementOf(CE_2))

DisjointClasses := 'DisjointClasses' '(' **axiomAnnotations ClassExpression ClassExpression** { **ClassExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

`DisjointClasses( a:Boy a:Girl )`	Nothing can be both a boy and a girl.
`ClassAssertion( a:Boy a:Stewie )`	Stewie is a boy.

The axioms in this ontology imply that *a:Stewie* can be classified as an instance of `ComplementOf( a:Girl )`. If the ontology were extended with the assertion `ClassAssertion( a:Girl a:Stewie )`, the ontology would become inconsistent.

9.1.4 Disjoint Union of Class Expressions

A disjoint union axiom DisjointUnion(C CE_1 ... CE_n) states that a class C is a disjoint union of the class expressions CE_i, $1 \leq i \leq n$, all of which are pairwise disjoint. Such axioms are sometimes referred to as *covering* axioms, as they state that the extensions of all CE_i exactly cover the extension of C. Thus, each instance of C is an instance of exactly one CE_i, and each instance of CE_i is an instance of C. Each such axiom can be seen as a syntactic shortcut for the following two axioms:

```
EquivalentClasses( C UnionOf( CE1 ... CEn ) )
DisjointClasses( CE1 ... CEn )
```

DisjointUnion := 'DisjointUnion' '(' **axiomAnnotations** **Class** **disjointClassExpressions** ')'
disjointClassExpressions := **ClassExpression** **ClassExpression** { **ClassExpression** }

Example:

Consider the ontology consisting of the following axioms.

`DisjointUnion( a:Child a:Boy a:Girl )`	Each child is either a boy or a girl, each boy is a child, each girl is a child, and nothing can be both a boy and a girl.
`ClassAssertion( a:Child a:Stewie )`	Stewie is a child.
`ClassAssertion( ComplementOf( a:Girl ) a:Stewie )`	Stewie is not a girl.

By the first two axioms, *a:Stewie* is either an instance of *a:Boy* or *a:Girl*. The last assertion eliminates the second possibility, so *a:Stewie* is classified as an instance of *a:Boy*.

9.2 Object Property Axioms

OWL 2 provides axioms that can be used to characterize and establish relationships between object property expressions. For clarity, the structure of these axioms is shown in two separate figures, Figure 14 and Figure 15. The **SubObjectPropertyOf** axiom allows one to state that the extension of one object property expression is included in the extension of another object property expression. The **EquivalentObjectProperties** axiom allows one to state that the extensions of several object property expressions are the same. The **DisjointObjectProperties** axiom allows one to state that the extensions of several object property expressions are pairwise disjoint — that is, that they do not share pairs of connected individuals. The **InverseObjectProperties** axiom can be used to state that two object property expressions are the inverse of each other. The **ObjectPropertyDomain** and **ObjectPropertyRange** axioms can be used to restrict the first and the second individual, respectively, connected by an object property expression to be instances of the specified class expression.

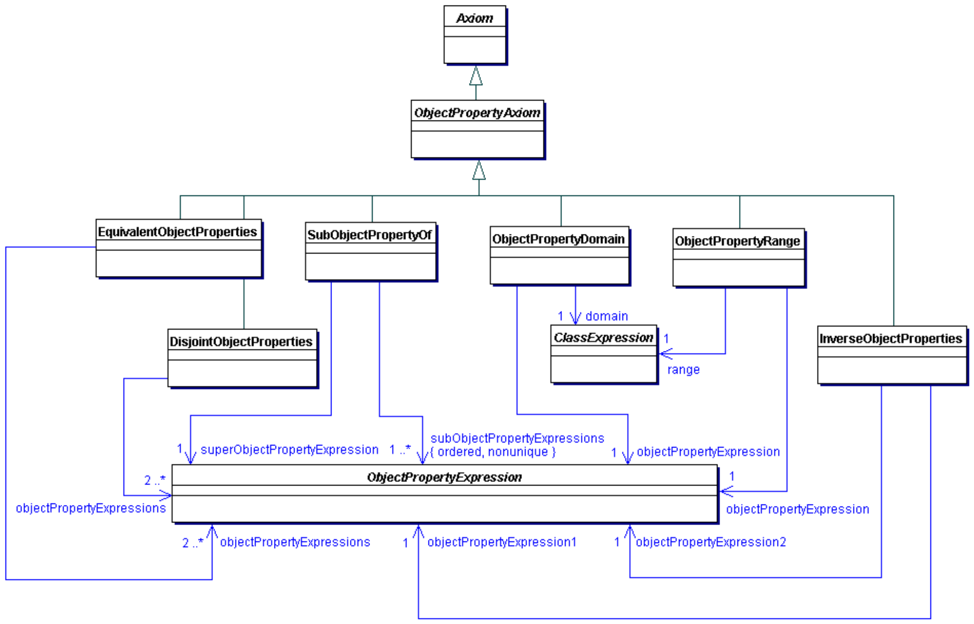
Axiom
ObjectPropertyAxiom
EquivalentObjectProperties
SubObjectPropertyOf
ObjectPropertyDomain
ObjectPropertyRange
DisjointObjectProperties
ClassExpression
InverseObjectProperties
1 domain
1 range
1 superObjectPropertyExpression
1 ..* subObjectPropertyExpressions { ordered, nonunique }
1 objectPropertyExpression
2 ..* objectPropertyExpressions
ObjectPropertyExpression
1 objectPropertyExpression
2 ..* objectPropertyExpressions
1 objectPropertyExpression1
1 objectPropertyExpression2

Figure 14. Object Property Axioms in OWL 2, Part I

The **FunctionalObjectProperty** axiom allows one to state that an object property expression is functional — that is, that each individual can have at most one outgoing connection of the specified object property expression. The

InverseFunctionalObjectProperty axiom allows one to state that an object property expression is inverse-functional — that is, that each individual can have at most one incoming connection of the specified object property expression. Finally, the **ReflexiveObjectProperty**, **IrreflexiveObjectProperty**, **SymmetricObjectProperty**, **AsymmetricObjectProperty**, and **TransitiveObjectProperty** axioms allow one to state that an object property expression is reflexive, irreflexive, symmetric, asymmetric, or transitive, respectively.

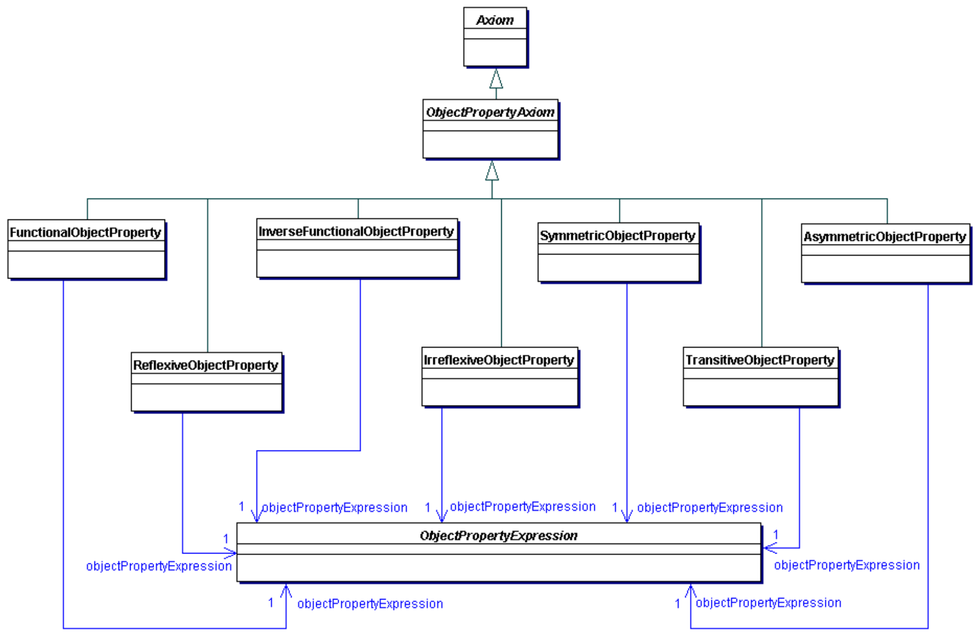

Figure 15. Axioms Defining Characteristics of Object Properties in OWL 2, Part II

ObjectPropertyAxiom :=
  SubObjectPropertyOf | **EquivalentObjectProperties** |
  DisjointObjectProperties | **InverseObjectProperties** |
  ObjectPropertyDomain | **ObjectPropertyRange** |
  FunctionalObjectProperty | **InverseFunctionalObjectProperty** |
  ReflexiveObjectProperty | **IrreflexiveObjectProperty** |
  SymmetricObjectProperty | **AsymmetricObjectProperty** |
  TransitiveObjectProperty

9.2.1 Object Subproperties

Object subproperty axioms are analogous to subclass axioms, and they come in two forms.

The basic form is `SubPropertyOf( OPE`$_1$ `OPE`$_2$ `)`. This axiom states that the object property expression `OPE`$_1$ is a subproperty of the object property expression `OPE`$_2$ — that is, if an individual `x` is connected by `OPE`$_1$ to an individual `y`, then `x` is also connected by `OPE`$_2$ to `y`.

The more complex form is `SubPropertyOf( PropertyChain( OPE`$_1$ `... OPE`$_n$ `) OPE )`. This axiom states that, if an individual `x` is connected by a sequence of object property expressions `OPE`$_1$, ..., `OPE`$_n$ with an individual `y`, then `x` is also connected with `y` by the object property expression `OPE`. Such axioms are also known as *complex role inclusions* [SROIQ].

SubObjectPropertyOf := 'SubPropertyOf' '(' **axiomAnnotations subObjectPropertyExpressions superObjectPropertyExpression** ')'
subObjectPropertyExpressions := **ObjectPropertyExpression** | **propertyExpressionChain**
propertyExpressionChain := 'PropertyChain' '(' **ObjectPropertyExpression ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'
superObjectPropertyExpression := **ObjectPropertyExpression**

Example:

Consider the ontology consisting of the following axioms.

`SubPropertyOf(` *a:hasDog a:hasPet* `)`	Having a dog is a kind of having a pet.
`PropertyAssertion(` *a:hasDog a:Peter a:Brian* `)`	Brian is a dog of Peter.

Since *a:hasDog* is a subproperty of *a:hasPet*, each tuple of individuals connected by the former property expression is also connected by the latter property expression. Therefore, this ontology entails that *a:Peter* is connected to *a:Brian* by *a:hasPet*; that is, the ontology entails the assertion `PropertyAssertion(` *a:hasPet a:Peter a:Brian* `)`.

Example:

Consider the ontology consisting of the following axioms.

`SubPropertyOf( PropertyChain(` *a:hasMother a:hasSister* `)` *a:hasAunt* `)`	The sister of someone's mother is that person's aunt.

`PropertyAssertion( a:hasMother a:Stewie a:Lois )`	Lois is the mother of Stewie.
`PropertyAssertion( a:hasSister a:Lois a:Carol )`	Carol is a sister of Lois.

The axioms in this ontology imply that *a:Stewie* is connected by *a:hasAunt* with *a:Carol*; that is, the ontology entails the assertion `PropertyAssertion( a:hasAunt a:Stewie a:Carol )`.

9.2.2 Equivalent Object Properties

An equivalent object properties axiom EquivalentProperties(OPE_1 ... OPE_n) states that all of the object property expressions OPE_i, 1 ≤ i ≤ n, are semantically equivalent to each other. This axiom allows one to use each OPE_i as a synonym for each OPE_j — that is, in any expression in the ontology containing such an axiom, OPE_i can be replaced with OPE_j without affecting the meaning of the ontology. The axiom EquivalentProperties(OPE_1 OPE_2) is equivalent to the following two axioms:

SubPropertyOf(OPE_1 OPE_2)
SubPropertyOf(OPE_2 OPE_1)

EquivalentObjectProperties := 'EquivalentProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

`EquivalentProperties( a:hasBrother a:hasMaleSibling )`	Having a brother is the same as having a male sibling.
`PropertyAssertion( a:hasBrother a:Chris a:Stewie )`	Stewie is a brother of Chris.
`PropertyAssertion( a:hasMaleSibling a:Stewie a:Chris )`	Chris is a male sibling of Stewie.

Since *a:hasBrother* and *a:hasMaleSibling* are equivalent properties, this ontology entails that *a:Chris* is connected by *a:hasMaleSibling* with *a:Stewie* — that is, the ontology entails the assertion `PropertyAssertion( a:hasMaleSibling` *a:Chris a:Stewie*) — and that *a:Stewie* is connected by *a:hasBrother* with *a:Chris* — that is, the ontology entails the assertion `PropertyAssertion(` *a:hasBrother a:Stewie a:Chris*).

9.2.3 Disjoint Object Properties

A disjoint object properties axiom `DisjointProperties(` OPE_1 ... OPE_n) states that all of the object property expressions OPE_i, 1 ≤ i ≤ n, are pairwise disjoint; that is, no individual `x` can be connected to an individual `y` by both OPE_i and OPE_j for i ≠ j.

DisjointObjectProperties := 'DisjointProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** { **ObjectPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

DisjointProperties(*a:hasFather a:hasMother*)	Fatherhood is disjoint with motherhood.
PropertyAssertion(*a:hasFather a:Stewie a:Peter*)	Peter is the father of Stewie.
PropertyAssertion(*a:hasMother a:Stewie a:Lois*)	Lois is the mother of Stewie.

In this ontology, the disjointness axiom is satisfied. If, however, one were to add an assertion `PropertyAssertion(` *a:hasMother a:Stewie a:Peter*), the disjointness axiom would be invalidated and the ontology would become inconsistent.

9.2.4 Inverse Object Properties

An inverse object properties axiom `InverseProperties(` OPE_1 OPE_2) states that the object property expression OPE_1 is an inverse of the object property expression OPE_2. Thus, if an individual `x` is connected by OPE_1 to an individual `y`, then `y` is also connected by OPE_2 to `x`, and vice versa. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
EquivalentProperties( OPE1 InverseOf( OPE2 ) )
```

InverseObjectProperties := 'InverseProperties' '(' **axiomAnnotations ObjectPropertyExpression ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`InverseProperties( a:hasFather a:fatherOf )`	Having a father is the opposite of being a father of someone.
`PropertyAssertion( a:hasFather a:Stewie a:Peter )`	Peter is the father of Stewie.
`PropertyAssertion( a:fatherOf a:Peter a:Chris )`	Peter is the father of Chris.

This ontology entails that *a:Peter* is connected by *a:fatherOf* with *a:Stewie* — that is, the ontology entails the assertion `PropertyAssertion( a:fatherOf a:Peter a:Stewie )` — and it also entails that *a:Chris* is connected by *a:hasFather* with *a:Peter* — that is, the ontology entails the assertion `PropertyAssertion( a:hasFather a:Chris a:Peter )`.

9.2.5 Object Property Domain

An object property domain axiom `PropertyDomain( OPE CE )` states that the domain of the object property expression `OPE` is the class expression `CE` — that is, if an individual `x` is connected by `OPE` with some other individual, then `x` is an instance of `CE`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( SomeValuesFrom( OPE owl:Thing ) CE )
```

ObjectPropertyDomain := 'PropertyDomain' '(' **axiomAnnotations ObjectPropertyExpression ClassExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyDomain( a:hasDog a:Person )`	Only people can own dogs.
`PropertyAssertion( a:hasDog a:Peter a:Brian )`	Brian is a dog of Peter.

By the first axiom, each individual that has an outgoing *a:hasDog* connection must be an instance of *a:Person*. Therefore, *a:Peter* can be classified as an instance of *a:Person*; that is, this ontology entails the assertion `ClassAssertion( a:Person a:Peter )`.

Domain axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. The domain axiom from the example ontology would in such systems be interpreted as a *constraint* saying that *a:hasDog* can point only from individuals that are known to be instances of *a:Person*; furthermore, since the example ontology does not explicitly state that *a:Peter* is an instance of *a:Person*, one might expect the domain constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is *inferred* from the domain constraint.

9.2.6 Object Property Range

An object property range axiom `PropertyRange( OPE CE )` states that the range of the object property expression `OPE` is the class expression `CE` — that is, if some individual is connected by `OPE` with an individual `x`, then `x` is an instance of `CE`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing AllValuesFrom( OPE CE ) )
```

ObjectPropertyRange := `'PropertyRange'` `'('` **axiomAnnotations** **ObjectPropertyExpression** **ClassExpression** `')'`

Example:

Consider the ontology consisting of the following axioms.

`PropertyRange( a:hasDog a:Dog )`	The range of the *a:hasDog* property is the class *a:Dog*.
`PropertyAssertion( a:hasDog a:Peter a:Brian )`	Brian is a dog of Peter.

By the first axiom, each individual that has an incoming *a:hasDog* connection must be an instance of *a:Dog*. Therefore, *a:Brian* can be classified as an instance of *a:Dog*; that is, this ontology entails the assertion `ClassAssertion( a:Brian a:Dog )`.

Range axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. The range axiom from the example ontology would in such systems be interpreted as a *constraint* saying that *a:hasDog* can point only to individuals that are known to be instances of *a:Dog*; furthermore, since the example ontology does not explicitly state that *a:Brian* is an instance of *a:Dog*, one might expect the range constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is *inferred* from the range constraint.

9.2.7 Functional Object Properties

An object property functionality axiom `FunctionalProperty( OPE )` states that the object property expression `OPE` is functional — that is, for each individual `x`, there can be at most one distinct individual `y` such that `x` is connected by `OPE` to `y`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing MaxCardinality( 1 OPE ) )
```

FunctionalObjectProperty := 'FunctionalProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`FunctionalProperty( a:hasFather )`	Each object can have at most one father.
`PropertyAssertion( a:hasFather a:Stewie a:Peter )`	Peter is the father of Stewie.
`PropertyAssertion( a:hasFather a:Stewie a:Peter_Griffin )`	Peter Griffin is the father of Stewie.

By the first axiom, *a:hasFather* can point from *a:Stewie* to at most one distinct individual, so *a:Peter* and *a:Peter_Griffin* must be equal; that is, this ontology entails the assertion `SameIndividual( a:Peter a:Peter_Griffin )`.

One might expect the previous ontology to be inconsistent, since the *a:hasFather* property points to two different values for *a:Stewie*. OWL 2, however, does not make the unique name assumption, so *a:Peter* and *a:Peter_Griffin* are not necessarily distinct individuals. If the ontology were extended with the axiom `DifferentIndividuals( a:Peter a:Peter_Griffin )`, then it would indeed become inconsistent.

9.2.8 Inverse-Functional Object Properties

An object property inverse functionality axiom `InverseFunctionalProperty( OPE )` states that the object property expression `OPE` is inverse-functional — that is, for each individual `x`, there can be at most one individual `y` such that `y` is connected by `OPE` with `x`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing MaxCardinality( 1 InverseOf( OPE ) )
)
```

InverseFunctionalObjectProperty := 'InverseFunctionalProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`InverseFunctionalProperty( a:fatherOf )`	Each object can have at most one father.
`PropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is the father of Stewie.
`PropertyAssertion( a:fatherOf a:Peter_Griffin a:Stewie )`	Peter Griffin is the father of Stewie.

By the first axiom, at most one distinct individual can point by *a:fatherOf* to *a:Stewie*, so *a:Peter* and *a:Peter_Griffin* must be equal; that is, this ontology entails the assertion `SameIndividual( a:Peter a:Peter_Griffin )`.

One might expect the previous ontology to be inconsistent, since there are two individuals that *a:Stewie* is connected to by *a:fatherOf*. OWL 2, however, does not make the unique name assumption, so *a:Peter* and *a:Peter_Griffin* are not necessarily distinct individuals. If the ontology were extended with the axiom

`DifferentIndividuals( a:Peter a:Peter_Griffin )`, then it would indeed become inconsistent.

9.2.9 Reflexive Object Properties

An object property reflexivity axiom `ReflexiveProperty( OPE )` states that the object property expression `OPE` is reflexive — that is, each individual is connected by `OPE` to itself.

ReflexiveObjectProperty := 'ReflexiveProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`ReflexiveProperty( a:knows )`	Everybody knows themselves.
`ClassAssertion( a:Person a:Peter )`	Peter is a person.

By the first axiom, *a:Peter* must be connected by *a:knows* to itself; that is, this ontology entails the assertion `PropertyAssertion( a:knows a:Peter a:Peter )`.

9.2.10 Irreflexive Object Properties

An object property irreflexivity axiom `IrreflexiveProperty( OPE )` states that the object property expression `OPE` is irreflexive — that is, no individual is connected by `OPE` to itself.

IrreflexiveObjectProperty := 'IrreflexiveProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`IrreflexiveProperty( a:marriedTo )`	Nobody can be married to themselves.

If this ontology were extended with the assertion `PropertyAssertion( a:marriedTo a:Peter a:Peter )`, the irreflexivity axiom would be contradicted and the ontology would become inconsistent.

9.2.11 Symmetric Object Properties

An object property symmetry axiom `SymmetricProperty( OPE )` states that the object property expression `OPE` is symmetric — that is, if an individual `x` is connected by `OPE` to an individual `y`, then `y` is also connected by `OPE` to `x`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubPropertyOf( OPE InverseOf( OPE ) )
```

SymmetricObjectProperty := `'SymmetricProperty'` `'('` **axiomAnnotations** **ObjectPropertyExpression** `')'`

Example:

Consider the ontology consisting of the following axioms.

`SymmetricProperty( a:friend )`	If x is a friend of y, they y is a friend of x.
`PropertyAssertion( a:friend a:Peter a:Brian )`	Brian is a friend of Peter.

Since *a:friend* is symmetric, *a:Peter* must be connected by *a:friend* to *a:Brian*; that is, this ontology entails the assertion `PropertyAssertion( a:friend a:Brian a:Peter )`.

9.2.12 Asymmetric Object Properties

An object property asymmetry axiom `AsymmetricProperty( OPE )` states that the object property expression `OPE` is asymmetric — that is, if an individual `x` is connected by `OPE` to an individual `y`, then `y` cannot be connected by `OPE` to `x`.

AsymmetricObjectProperty := 'AsymmetricProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`AsymmetricProperty( a:parentOf )`	If x is a parent of y, they y is not a parent of x.
`PropertyAssertion( a:parentOf a:Peter a:Stewie )`	Peter is a parent of Stewie.

If this ontology were extended with the assertion `PropertyAssertion( a:parentOf a:Stewie a:Peter )`, the asymmetry axiom would be invalidated and the ontology would become inconsistent.

9.2.13 Transitive Object Properties

An object property transitivity axiom `TransitiveProperty( OPE )` states that the object property expression `OPE` is transitive — that is, if an individual `x` is connected by `OPE` to an individual `y` that is connected by `OPE` to an individual `z`, then `x` is also connected by `OPE` to `z`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubPropertyOf( PropertyChain( OPE OPE ) OPE )
```

TransitiveObjectProperty := 'TransitiveProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`TransitiveProperty( a:ancestorOf )`	If x is an ancestor of y and y is an ancestor of z, then x is an ancestor of z.
`PropertyAssertion( a:ancestorOf a:Carter a:Lois )`	Carter is an ancestor of Lois.
`PropertyAssertion( a:ancestorOf a:Lois a:Meg )`	Lois is an ancestor of Meg.

Since *a:ancestorOf* is transitive, *a:Carter* must be connected by *a:ancestorOf* to *a:Meg*; that is, this ontology entails the assertion `PropertyAssertion( a:ancestorOf a:Carter a:Meg ).`

9.3 Data Property Axioms

OWL 2 also provides for data property axioms. Their structure is similar to object property axioms, as shown in Figure 16. The **SubDataPropertyOf** axiom allows one to state that the extension of one data property expression is included in the extension of another data property expression. The **EquivalentDataProperties** allows one to state that several data property expressions have the same extension. The **DisjointDataProperties** axiom allows one to state that the extensions of several data property expressions are disjoint with each other — that is, they do not share individual–literal pairs. The **DataPropertyDomain** axiom can be used to restrict individuals connected by a property expression to be instances of the specified class; similarly, the **DataPropertyRange** axiom can be used to restrict the literals pointed to by a property expression to be in the specified unary data range. Finally, the **FunctionalDataProperty** axiom allows one to state that a data property expression is functional — that is, that each individual can have at most one outgoing connection of the specified data property expression.

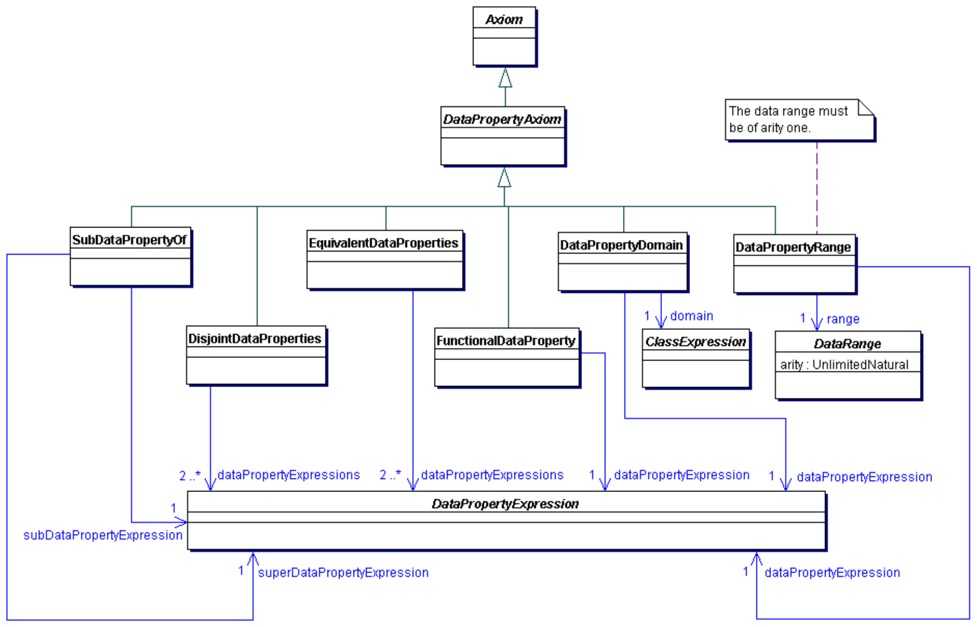

Figure 16. Data Property Axioms of OWL 2

DataPropertyAxiom :=
    SubDataPropertyOf | **EquivalentDataProperties** | **DisjointDataProperties** |
    DataPropertyDomain | **DataPropertyRange** | **FunctionalDataProperty**

9.3.1 Data Subproperties

A data subproperty axiom `SubPropertyOf(` DPE_1 DPE_2 `)` states that the data property expression DPE_1 is a subproperty of the data property expression DPE_2 — that is, if an individual `x` is connected by OPE_1 to a literal `y`, then `x` is connected by OPE_2 to `y` as well.

SubDataPropertyOf := `'SubPropertyOf'` `'('` **axiomAnnotations** **subDataPropertyExpression** **superDataPropertyExpression** `')'`
subDataPropertyExpression := **DataPropertyExpression**
superDataPropertyExpression := **DataPropertyExpression**

Example:

Consider the ontology consisting of the following axioms.

`SubPropertyOf(` *a:hasLastName* *a:hasName* `)`	Having a last name is a kind of having a name.
`PropertyAssertion(` *a:hasLastName* *a:Peter* `"Griffin" )`	Peter's last name is `"Griffin"`.

Since *a:hasLastName* is a subproperty of *a:hasName*, each individual connected by the former property to a literal is also connected by the latter property to the same literal. Therefore, this ontology entails that *a:Peter* is connected to `"Peter"` through *a:hasName*; that is, the ontology entails the assertion `PropertyAssertion(` *a:hasName* *a:Peter* `"Peter" )`.

9.3.2 Equivalent Data Properties

An equivalent data properties axiom `EquivalentProperties(` DPE_1 ... DPE_n `)` states that all the data property expressions DPE_i, 1 ≤ i ≤ n, are semantically equivalent to each other. This axiom allows one to use each DPE_i as a synonym for each DPE_j — that is, in any expression in the ontology containing such an axiom, DPE_i can be replaced with DPE_j without affecting the meaning of the ontology. The axiom `EquivalentProperties( DPE1 DPE2 )` can be seen as a syntactic shortcut for the following axiom:

```
SubPropertyOf( DPE1 DPE2 )
SubPropertyOf( DPE2 DPE1 )
```

EquivalentDataProperties := 'EquivalentProperties' '(' **axiomAnnotations** **DataPropertyExpression** **DataPropertyExpression** { **DataPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

EquivalentProperties(*a:hasName* *a:seLlama*)	*a:hasName* and *a:seLlama* (in Spanish) are synonyms.
PropertyAssertion(*a:hasName* *a:Meg* "Meg Griffin")	Meg's name is "Meg Griffin".
PropertyAssertion(*a:seLlama* *a:Meg* "Megan Griffin")	Meg's name is "Megan Griffin".

Since *a:hasName* and *a:seLlama* are equivalent properties, this ontology entails that *a:Meg* is connected by *a:seLlama* with "Meg Griffin" — that is, the ontology entails the assertion PropertyAssertion(*a:seLlama* *a:Meg* "Meg Griffin") — and that *a:Meg* is also connected by *a:hasName* with "Megan Griffin" — that is, the ontology entails the assertion PropertyAssertion(*a:hasName* *a:Meg* "Megan Griffin").

9.3.3 Disjoint Data Properties

A disjoint data properties axiom DisjointProperties(DPE_1 ... DPE_n) states that all of the data property expressions DPE_i, 1 ≤ i ≤ n, are pairwise disjoint; that is, no individual x can be connected to a literal y by both DPE_i and DPE_j for i ≠ j.

DisjointDataProperties := 'DisjointProperties' '(' **axiomAnnotations** **DataPropertyExpression** **DataPropertyExpression** { **DataPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

`DisjointProperties( a:hasName a:hasAddress )`	Someone's name must be different from his address.
`PropertyAssertion( a:hasName a:Peter "Peter Griffin" )`	Peter's name is `"Peter Griffin"`.
`PropertyAssertion( a:hasAddress a:Peter "Quahog, Rhode Island" )`	Peter's address is `"Quahog, Rhode Island"`.

In this ontology, the disjointness axiom is satisfied. If, however, one were to add an assertion `PropertyAssertion( a:hasAddress a:Peter "Peter Griffin" )`, the disjointness axiom would be invalidated and the ontology would become inconsistent.

9.3.4 Data Property Domain

A data property domain axiom `PropertyDomain( DPE CE )` states that the domain of the data property expression `DPE` is the class expression `CE` — that is, if an individual `x` is connected by `DPE` with some literal, then `x` is an instance of `CE`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( SomeValuesFrom( DPE rdfs:Literal) CE )
```

DataPropertyDomain := `'PropertyDomain'` `'('` **axiomAnnotations** **DataPropertyExpression** **ClassExpression** `')'`

Example:

Consider the ontology consisting of the following axioms.

`PropertyDomain( a:hasName a:Person )`	Only people can have names.
`PropertyAssertion( a:hasName a:Peter "Peter Griffin" )`	Peter's name is `"Peter Griffin"`.

By the first axiom, each individual that has an outgoing *a:hasName* connection must be an instance of *a:Person*. Therefore, *a:Peter* can be classified as an instance of *a:Person*; that is, this ontology entails the assertion `ClassAssertion( a:Person a:Peter )`.

Domain axioms in OWL 2 have a standard first-order semantics that is somewhat different from the semantics of such axioms in databases and object-oriented systems, where such axioms are interpreted as checks. Thus, the domain axiom from the example ontology would in such systems be interpreted as a *constraint* saying that *a:hasName* can point only from individuals that are known to be instances of *a:Person*; furthermore, since the example ontology does not explicitly state that *a:Peter* is an instance of *a:Person*, one might expect the domain constraint to be invalidated. This, however, is not the case in OWL 2: as shown in the previous paragraph, the missing type is *inferred* from the domain constraint.

9.3.5 Data Property Range

A data property range axiom `PropertyRange( DPE DR )` states that the range of the data property expression `DPE` is the data range `DR` — that is, if some individual is connected by `DPE` with a literal `x`, then `x` is in `DR`. The arity of `DR` *must* be one. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing AllValuesFrom( DPE DR ) )
```

DataPropertyRange := `'PropertyRange'` `'('` **axiomAnnotations** **DataPropertyExpression** **DataRange** `')'`

Example:

Consider the ontology consisting of the following axioms.

`PropertyRange( a:hasName xsd:string )`	The range of the *a:hasName* property is *xsd:string*.
`PropertyAssertion( a:hasName a:Peter "Peter Griffin" )`	Peter's name is "`Peter Griffin`".

By the first axiom, each literal that has an incoming *a:hasName* link must be in *xsd:string*. In the example ontology, this axiom is satisfied. If, however, the ontology were extended with an assertion `PropertyAssertion( a:hasName a:Peter "42"^^xsd:integer )`, the range axiom would imply that the literal `"42"^^xsd:integer` is in *xsd:string*, which is a contradiction; therefore, the ontology would become inconsistent.

9.3.6 Functional Data Properties

A data property functionality axiom `FunctionalProperty( DPE )` states that the data property expression `DPE` is functional — that is, for each individual `x`, there can be at most one distinct literal `y` such that `x` is connected by `DPE` with `y`. Each such axiom can be seen as a syntactic shortcut for the following axiom:

```
SubClassOf( owl:Thing MaxCardinality( 1 DPE ) )
```

FunctionalDataProperty := 'FunctionalProperty' '(' **axiomAnnotations** **DataPropertyExpression** ')'

Example:

Consider the ontology consisting of the following axioms.

`FunctionalProperty( a:hasAge )`	Each object can have at most one age.
`PropertyAssertion( a:hasAge a:Meg "17"^^xsd:integer )`	Meg is seventeen years old.

By the first axiom, *a:hasAge* can point from *a:Meg* to at most one distinct literal. In this example ontology, this axiom is satisfied. If, however, the ontology were extended with the assertion `PropertyAssertion( a:hasAge a:Meg "15"^^xsd:integer )`, the semantics of functionality axioms would imply that `"15"^^xsd:integer` is equal to `"17"^^xsd:integer`, which is a contradiction; therefore, the ontology would become inconsistent.

9.4 Keys

A key axiom `HasKey( CE PE`$_1$` ... PE`$_n$` )` states that each (named) instance of the class expression `CE` is uniquely identified by the (data or object) property expressions `PE`$_i$ — that is, no two distinct (named) instances of `CE` can coincide on the values of all property expressions `PE`$_i$. A key axiom of the form `HasKey( owl:Thing OPE )` is similar to the axiom `InverseFunctionalProperty( OPE )`; the main difference is that the first axiom is applicable only to individuals that are explicitly named in an ontology, while the second axiom is also applicable to individuals whose existence is implied by existential quantification. The structure of such axiom is shown in Figure 17.

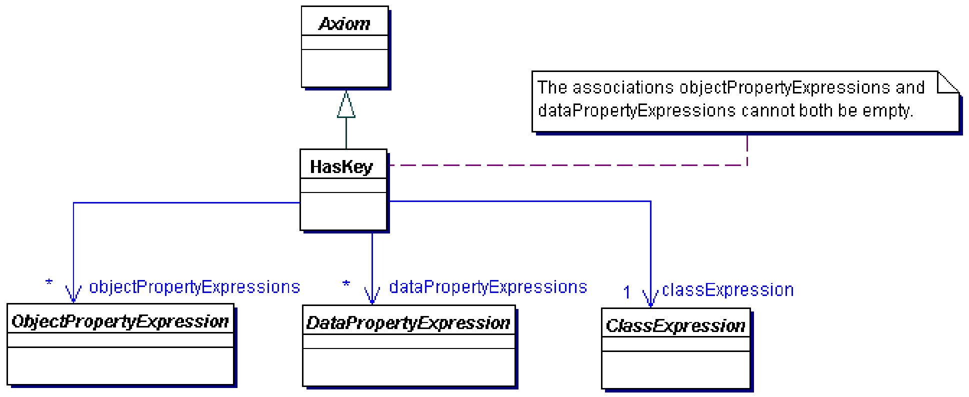
Axiom
The associations objectPropertyExpressions and dataPropertyExpressions cannot both be empty.
HasKey
* objectPropertyExpressions
* dataPropertyExpressions
1 classExpression
ObjectPropertyExpression
DataPropertyExpression
ClassExpression

Figure 17. Key Axioms in OWL 2

HasKey := 'HasKey' '(' **axiomAnnotations** **ClassExpression** **ObjectPropertyExpression** | **DataPropertyExpression** { **ObjectPropertyExpression** | **DataPropertyExpression** } ')'

Example:

Consider the ontology consisting of the following axioms.

`HasKey( a:Person a:hasSSN )`	Each person is uniquely identified by their social security number.
`PropertyAssertion( a:hasSSN a:Peter "123-45-6789" )`	Peter's social security number is `"123-45-6789"`.
`ClassAssertion( a:Person a:Peter )`	Peter is a person.
`PropertyAssertion( a:hasSSN a:Peter_Griffin "123-45-6789" )`	Peter Griffin's social security number is `"123-45-6789"`.
`ClassAssertion( a:Person a:Peter_Griffin )`	Peter Griffin is a person.

The first axiom makes *a:hasSSN* the key for individuals in the class *a:Person*; thus, if an instance of *a:Person* has a value for *a:hasSSN*, then this value must be unique. Since the values of *a:hasSSN* are the same for *a:Peter* and *a:Peter_Griffin*, these two individuals must be equal — that is, this ontology entails the assertion `SameIndividual( a:Peter a:Peter_Griffin )`.

One might expect the previous ontology to be inconsistent, since the *a:hasSSN* has the same value for two individuals *a:Peter* and *a:Peter_Griffin*. However, OWL 2 does not make the unique name assumption, so *a:Peter* and *a:Peter_Griffin* are not necessarily distinct individuals. If the ontology were extended with the axiom `DifferentIndividuals( a:Peter a:Peter_Griffin )`, then it would indeed become inconsistent.

The semantics of key axioms is specific in that these axioms apply only to individuals explicitly introduced in the ontology by name, and not to unnamed individuals (i.e., the individuals whose existence is implied by existential quantification). This makes key axioms equivalent to a variant of DL-safe rules [DL-Safe]. Thus, key axioms will typically not affect class-based inferences such as the computation of the subsumption hierarchy, but they will play a role in answering queries about individuals. This choice has been made in order to keep the language decidable.

Example:

Consider the ontology consisting of the following axioms.

`HasKey( a:Person a:hasSSN )`	Each person is uniquely identified by their social security number.
`PropertyAssertion( a:hasSSN a:Peter "123-45-6789" )`	Peter's social security number `"123-45-6789"`.
`ClassAssertion( a:Person a:Peter )`	Peter is a person.
`ClassAssertion( SomeValuesFrom( a:marriedTo IntersectionOf( a:Man HasValue( a:hasSSN "123-45-6789" ) ) ) a:Lois )`	Lois is married to some man whose social security number is `"123-45-6789"`.
`SubClassOf( a:Man a:Person )`	Each man is a person.

The fourth axiom implies existence of some individual `x` that is an instance of *a:Man* and whose value for the *a:hasSSN* data property is `"123-45-6789"`; by the fifth axiom, `x` is an instance of *a:Person* as well. Furthermore, the second and the third axiom say that *a:Peter* is an instance of *a:Person* and that the value of *a:hasSSN* for *a:Peter* is `"123-45-6789"`. Finally, the first axiom says that *a:hasSSN* is a key property for instances of *a:Person*. Thus, one might expect `x` to be equal to *a:Peter*, and for the ontology to entail the assertion `ClassAssertion( a:Man a:Peter )`.

The inferences in the previous paragraph, however, cannot be drawn because of the DL-safe semantics of key axioms: `x` is an individual that has not been explicitly named in the ontology; therefore, the semantics of key axioms does not apply to `x`. Therefore, this OWL 2 ontology does not entail the assertion `ClassAssertion( a:Man a:Peter )`.

9.5 Assertions

OWL 2 supports a rich set of axioms for stating *assertions* — axioms about individuals that are often also called *facts*. For clarity, different types of assertions are shown in three separate figures, Figure 18, 19, and 20. The **SameIndividual** assertion allows one to state that several individuals are all equal to each other, while the **DifferentIndividuals** assertion allows for the opposite — that is, to state that several individuals are all different from each other. The **ClassAssertion** axiom allows one to state that an individual is an instance of a particular class.

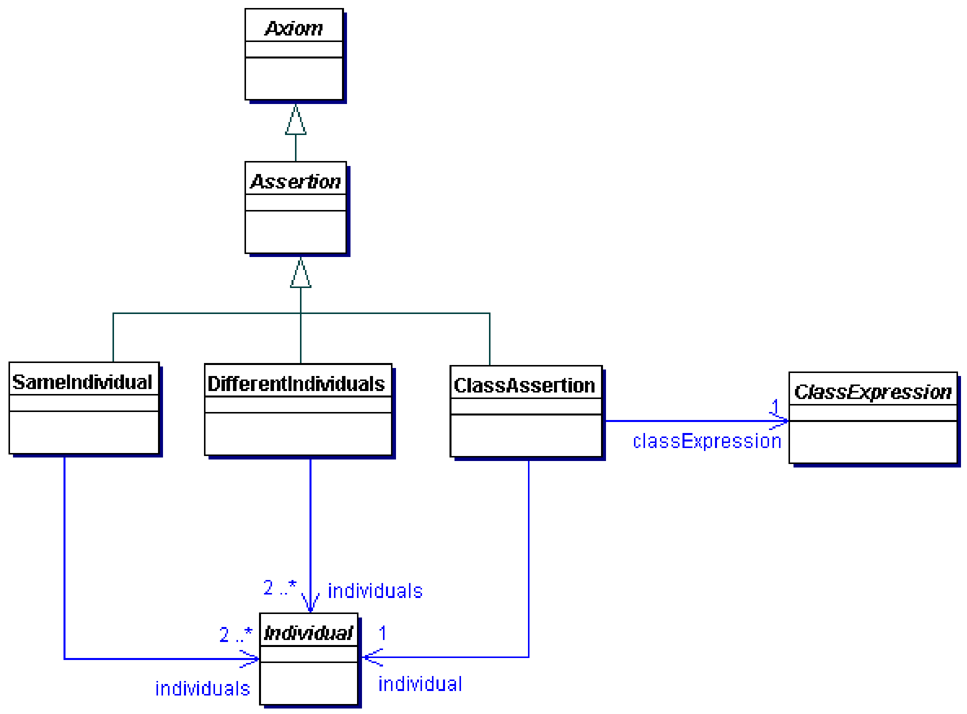

Figure 18. Class and Individual (In)Equality Assertions in OWL 2

The **ObjectPropertyAssertion** axiom allows one to state that an individual is connected by an object property expression to an individual, while **NegativeObjectPropertyAssertion** allows for the opposite — that is, to state that an individual is not connected by an object property expression to an individual.

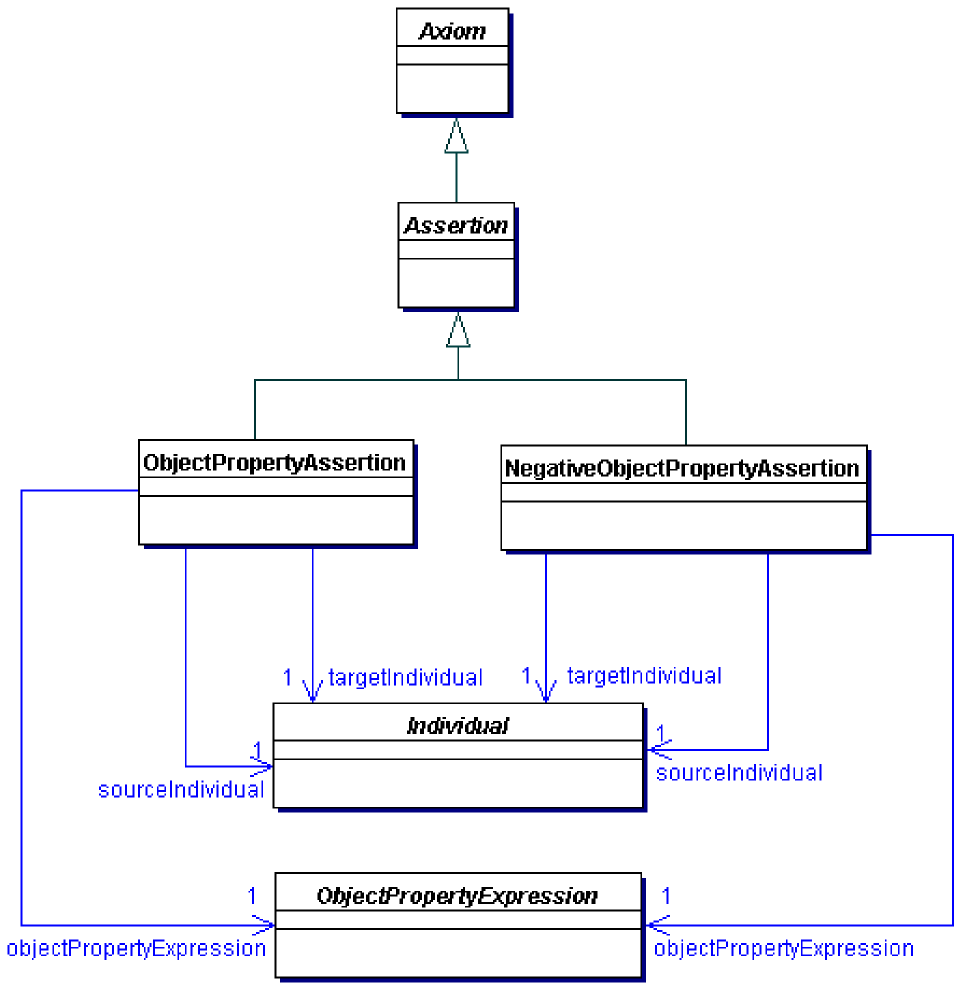

Figure 19. Object Property Assertions in OWL 2

The **DataPropertyAssertion** axiom allows one to state that an individual is connected by a data property expression to literal, while **NegativeDataPropertyAssertion** allows for the opposite — that is, to state that an individual is not connected by a data property expression to a literal.

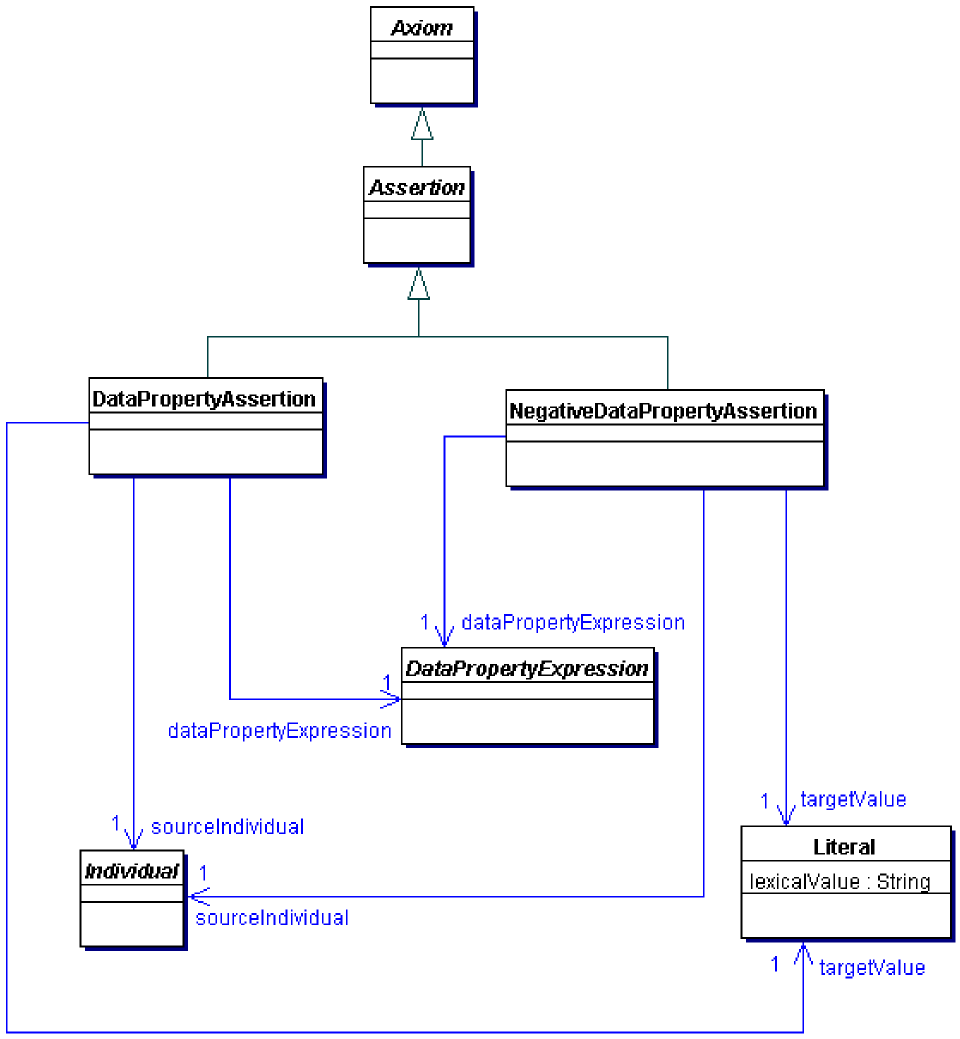

Figure 20. Data Property Assertions in OWL 2

Assertion :=
  SameIndividual | **DifferentIndividuals** | **ClassAssertion** |
  ObjectPropertyAssertion | **NegativeObjectPropertyAssertion** |
  DataPropertyAssertion | **NegativeDataPropertyAssertion**

sourceIndividual := **Individual**
targetIndividual := **Individual**
targetValue := **Literal**

9.5.1 Individual Equality

An individual equality axiom `SameIndividual( a1 ... an )` states that all of the individuals a_i, 1 ≤ i ≤ n, are equal to each other. This axiom allows one to use each a_i as a synonym for each a_j — that is, in any expression in the ontology containing such an axiom, a_i can be replaced with a_j without affecting the meaning of the ontology.

SameIndividual := 'SameIndividual' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

Example:

Consider the ontology consisting of the following axioms.

SameIndividual(*a:Meg a:Megan*)	Meg and Megan are the same objects.
PropertyAssertion(*a:hasBrother a:Meg a:Stewie*)	Meg has a brother Stewie.

Since *a:Meg* and *a:Megan* are equal, one individual can always be replaced with the other one. Therefore, this ontology entails that *a:Megan* is connected by *a:hasBrother* with *a:Stewie* — that is, the ontology entails the assertion PropertyAssertion(*a:hasBrother a:Megan a:Stewie*).

9.5.2 Individual Inequality

An individual inequality axiom `DifferentIndividuals( a1 ... an )` states that all of the individuals a_i, 1 ≤ i ≤ n, are different from each other; that is, no individuals a_i and a_j with i ≠ j can be derived to be equal. This axiom can be used to axiomatize the *unique name assumption* — the assumption that all different individual names denote different individuals.

DifferentIndividuals := 'DifferentIndividuals' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:fatherOf a:Peter a:Meg )`	Peter is the father of Meg.
`PropertyAssertion( a:fatherOf a:Peter a:Chris )`	Peter is the father of Chris.
`PropertyAssertion( a:fatherOf a:Peter a:Stewie )`	Peter is the father of Stewie.
`DifferentIndividuals( a:Peter a:Meg a:Chris a:Stewie )`	Peter, Meg, Chris, and Stewie are all different from each other.

The last axiom in this example ontology axiomatizes the unique name assumption (but only for the three names in the axiom). If the ontology were extended with an axiom `FunctionalProperty( a:fatherOf )`, this axiom would imply that *a:Meg*, *a:Chris*, and *a:Stewie* are all equal, which would invalidate the unique name assumption and would make the ontology inconsistent.

9.5.3 Class Assertions

A class assertion `ClassAssertion( CE a )` states that the individual `a` is an instance of the class expression `CE`.

ClassAssertion `:=` `'ClassAssertion'` `'('` **axiomAnnotations** **ClassExpression** **Individual** `')'`

Example:

Consider the ontology consisting of the following axioms.

`ClassAssertion( a:Dog a:Brian )`	Brian is a dog.
`SubClassOf( a:Dog a:Mammal )`	Each dog is a mammal.

The first axiom states that *a:Brian* is an instance of the class *a:Dog*. By the second axiom, each instance of *a:Dog* is an instance of *a:Mammal*. Therefore, this ontology entails that *a:Brian* is an instance of *a:Mammal* — that is, the ontology entails the assertion `ClassAssertion( a:Mammal a:Brian )`.

9.5.4 Positive Object Property Assertions

A positive object property assertion `PropertyAssertion( OPE a1 a2 )` states that the individual a_1 is connected by the object property expression `OPE` to the individual a_2.

ObjectPropertyAssertion := 'PropertyAssertion' '(' **axiomAnnotations** **ObjectPropertyExpression** **sourceIndividual** **targetIndividual** ')'

Example:

Consider the ontology consisting of the following axioms.

`PropertyAssertion( a:hasDog a:Peter a:Brian )`	Brian is a dog of Peter.
`SubClassOf( SomeValuesFrom( a:hasDog owl:Thing ) a:DogOwner )`	Things having a dog are dog owners.

The first axiom states that *a:Peter* is connected by *a:hasDog* to *a:Brian*. By the second axioms, each individual connected by *a:hasDog* to an individual is an instance of *a:DogOwner*. Therefore, this ontology entails that *a:Peter* is an instance of *a:DogOwner* — that is, the ontology entails the assertion `ClassAssertion( a:DogOwner a:Peter )`.

9.5.5 Negative Object Property Assertions

A negative object property assertion `NegativePropertyAssertion( OPE a1 a2 )` states that the individual a_1 is not connected by the object property expression `OPE` to the individual a_2.

NegativeObjectPropertyAssertion := 'NegativePropertyAssertion' '(' **axiomAnnotations** **objectPropertyExpression** **sourceIndividual** **targetIndividual** ')'

Example:

Consider the ontology consisting of the following axiom.

```
NegativePropertyAssertion(
a:hasSon a:Peter a:Meg )
```

Meg is not a son of Peter.

If this ontology were extended with an assertion `PropertyAssertion( a:hasSon a:Peter a:Meg )`, the negative object property assertion would be invalidated and the ontology would become inconsistent.

9.5.6 Positive Data Property Assertions

A positive data property assertion `PropertyAssertion( DPE a lt )` states that the individual `a` is connected by the data property expression `DPE` to the literal `lt`.

DataPropertyAssertion := 'PropertyAssertion' '(' **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** ')'

Example:

Consider the ontology consisting of the following axioms.

```
PropertyAssertion( a:hasAge
a:Meg "17"^^xsd:integer )
```

Meg is seventeen years old.

```
SubClassOf(
    SomeValuesFrom( a:hasAge
       DatatypeRestriction(
xsd:integer
          xsd:minInclusive
"13"^^xsd:integer
          xsd:maxInclusive
"19"^^xsd:integer
       )
    )
    a:Teenager
)
```

Things older than 13 and younger than 19 (both inclusive) are teenagers.

The first axiom states that *a:Meg* is connected by *a:hasAge* to the literal `"17"^^xsd:integer`. By the second axioms, each individual connected by *a:hasAge* to an integer between 13 and 19 is an instance of *a:Teenager*. Therefore, this ontology entails that *a:Meg* is an instance of *a:Teenager* — that is, the ontology entails the assertion `ClassAssertion( a:Teenager a:Meg )`.

9.5.7 Negative Data Property Assertions

A negative data property assertion `NegativePropertyAssertion( DPE a lt )` states that the individual `a` is not connected by the data property expression `DPE` to the literal `lt`.

NegativeDataPropertyAssertion := `'NegativePropertyAssertion'` `'('` **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** `')'`

Example:

Consider the ontology consisting of the following axiom.

```
NegativePropertyAssertion(
a:hasAge a:Meg "5"^^xsd:integer     Meg is not five years old.
)
```

If this ontology were extended with an assertion `PropertyAssertion( a:hasAge a:Meg "5"^^xsd:integer )`, the negative data property assertion would be invalidated and the ontology would become inconsistent.

10 Annotations

OWL 2 applications often need ways to associate information with ontologies, entities, and axioms in a way that does not affect the logical meaning of the ontology. Such information often plays a central role in OWL 2 applications. Although such information does not affect the formal meaning of an ontology (i.e., it does not affect the set of logical consequences that one can derive from an ontology), it is expected to be accessible in the structural specification of OWL 2. To this end, OWL 2 provides for *annotations* on ontologies, axioms, and entities.

Example:

One might want to associate human-readable labels with IRIs and use them when visualizing an ontology. To this end, one might use the *rdfs:label* annotation property to associate such labels with ontology IRIs.

Various OWL 2 syntaxes, such as the functional-style syntax, provide a mechanism for embedding comments into ontology documents. The structure of such comments is, however, dependent on the syntax, so these are simply discarded during parsing. In contrast, annotations are "first-class citizens" in the structural specification of OWL 2, and their structure is independent of the underlying syntax.

Example:

Since it is based on XML, the OWL 2 XML Syntax [OWL 2 XML Syntax] allows the embedding of the standard XML comments into ontology documents. Such comments are not represented in the structural specification of OWL 2 and, consequently, they should be ignored during document parsing.

10.1 Annotations of Ontologies, Axioms, and other Annotations

Ontologies, axioms, and annotations themselves can be annotated using annotations shown in Figure 21. As shown in the figure, such annotations consist of an annotation property and an annotation value, where the latter can be anonymous individuals, IRIs, and literals.

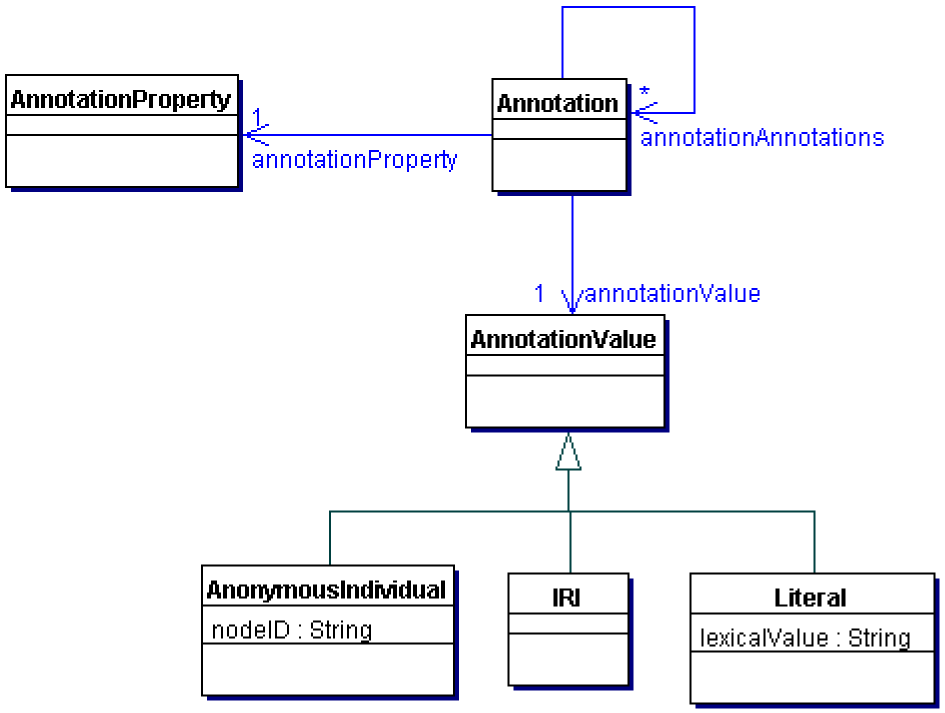

Figure 21. Annotations of Ontologies and Axioms in OWL 2

Annotation := 'Annotation' '(' **annotationAnnotations** **AnnotationProperty** **AnnotationValue** ')'

annotationAnnotations := { **Annotation** }
AnnotationValue := **AnonymousIndividual** | **IRI** | **Literal**

10.2 Annotation Axioms

OWL 2 provides means to state several types of axioms about annotation properties, as shown in Figure 22. These axioms have no effect on the Direct Semantics of OWL 2 [*OWL 2 Direct Semantics*], and they are treated as axioms only in order to simplify the structural specification of OWL 2.

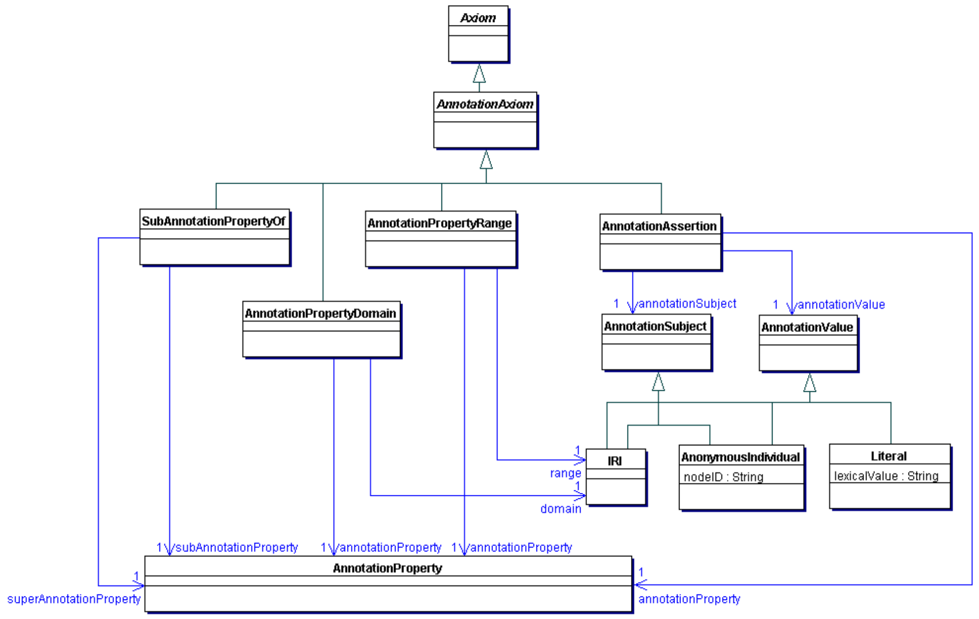

Figure 22. Annotations of IRIs and Anonymous Individuals in OWL 2

AnnotationAxiom := **AnnotationAssertion** | **SubAnnotationPropertyOf** | **AnnotationPropertyDomain** | **AnnotationPropertyRange**

10.2.1 Annotation Assertion

An annotation assertion `AnnotationAssertion( AP as at )` states that the annotation subject `as` — an IRI or an anonymous individual — is annotated with the annotation property `AP` and the annotation value `av`. Such axioms have no effect on the Direct Semantics of OWL 2 [*OWL 2 Direct Semantics*].

AnnotationAssertion := 'AnnotationAssertion' '(' **axiomAnnotations** **AnnotationProperty** **AnnotationSubject** **AnnotationValue** ')'
AnnotationSubject := **IRI** | **AnonymousIndividual**

Example:

The following axiom assigns a human-readable comment to the IRI *a:Person*.

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

Since the annotation is assigned to an IRI, it applies to all entities with the given IRI. Thus, if an ontology contains both a class and an individual *a:Person*, the above comment applies to both entities.

10.2.2 Annotation Subproperties

An annotation subproperty axiom `SubPropertyOf( AP`$_1$ `AP`$_2$ `)` states that the annotation property `AP`$_1$ is a subproperty of the annotation property `AP`$_2$. Such axioms have no effect on the Direct Semantics of OWL 2 [*OWL 2 Direct Semantics*].

SubAnnotationPropertyOf := 'SubPropertyOf' '(' **axiomAnnotations** **subAnnotationProperty** **superAnnotationProperty** ')'
subAnnotationProperty := **AnnotationProperty**
superAnnotationProperty := **AnnotationProperty**

10.2.3 Annotation Property Domain

An annotation property domain axiom `PropertyDomain( AP U )` states that the domain of the annotation property `AP` is the IRI `U`. Such axioms have no effect on the Direct Semantics of OWL 2 [*OWL 2 Direct Semantics*].

AnnotationPropertyDomain := 'PropertyDomain' '(' **axiomAnnotations** **AnnotationProperty** **IRI** ')'

10.2.4 Annotation Property Range

An annotation property range axiom `PropertyRange( AP U )` states that the range of the annotation property `AP` is the IRI `U`. Such axioms have no effect on the Direct Semantics of OWL 2 [OWL 2 Direct Semantics].

AnnotationPropertyRange := 'PropertyRange' '(' **axiomAnnotations** **AnnotationProperty** **IRI** ')'

11 Global Restrictions on Axioms

The axiom closure *Ax* (with anonymous individuals renamed apart as explained in Section 5.6.2) of each OWL 2 ontology *O must* satisfy the *global restrictions* defined in this section. As explained in the literature [*SROIQ*], this restriction is necessary in order to obtain a decidable language. The formal definition of these conditions is rather technical, so it is split into two parts. Section 11.1 first introduces the notions of a property hierarchy and of *simple* object property expressions. These notions are then used in Section 11.2 to define the actual conditions on *Ax*.

11.1 Property Hierarchy and Simple Object Property Expressions

For an object property expression `OPE`, the *inverse property expression* `INV(OPE)` is defined as follows:

- If `OPE` is an object property `OP`, then `INV(OPE)` = `InverseOf( OP )`.
- if `OPE` is of the form `InverseOf( OP )` for `OP` an object property, then `INV(OPE)` = `OP`.

The set *AllOPE(Ax)* of all object property expressions w.r.t. *Ax* is the smallest set containing `OP` and `INV(OP)` for each object property `OP` occurring in *Ax*.

An object property expression `OPE` is *composite* in the set of axioms *Ax* if

- `OPE` is equal to *owl:topObjectProperty* or *owl:bottomObjectProperty*, or
- *Ax* contains an axiom of the form
  - `SubPropertyOf( PropertyChain(` OPE_1 ... OPE_n `) OPE )` with n > 1, or
  - `SubPropertyOf( PropertyChain(` OPE_1 ... OPE_n `) INV(OPE) )` with n > 1, or
  - `TransitiveProperty( OPE )`, or
  - `TransitiveProperty( INV(OPE) )`.

The relation → is the smallest relation on *AllOPE(Ax)* for which the following conditions hold (`A` → `B` means that → holds for `A` and `B`):

- if *Ax* contains an axiom `SubPropertyOf(` OPE_1 OPE_2 `)`, then $OPE_1 \rightarrow OPE_2$ holds; and
- if *Ax* contains an axiom `EquivalentProperties(` OPE_1 OPE_2 `)`, then $OPE_1 \rightarrow OPE_2$ and $OPE_2 \rightarrow OPE_1$ hold; and
- if *Ax* contains an axiom `InverseProperties(` OPE_1 OPE_2 `)`, then $OPE_1 \rightarrow INV(OPE_2)$ and $INV(OPE_2) \rightarrow OPE_1$ hold; and
- if *Ax* contains an axiom `SymmetricProperty(OPE)`, then `OPE` → `INV(OPE)` holds; and
- if $OPE_1 \rightarrow OPE_2$ holds, then $INV(OPE_1) \rightarrow INV(OPE_2)$ holds as well.

The *property hierarchy* relation $\rightarrow^*$ is the reflexive-transitive closure of →.

An object property expression `OPE` is *simple* in *Ax* if, for each object property expression `OPE'` such that `OPE'` $\rightarrow^*$ `OPE` holds, `OPE'` is not composite.

Example:

Roughly speaking, a simple object property expression has no direct or indirect subproperties that are either transitive or are defined by means of property chains, where the notion of indirect subproperties is captured by the property hierarchy. Consider the following axioms:

`SubPropertyOf( PropertyChain( a:hasFather a:hasBrother ) a:hasUncle )`	The brother of someone's father is that person's uncle.
`SubPropertyOf( a:hasUncle a:hasRelative )`	Having an uncle is a kind of having a relative.
`SubPropertyOf( a:hasBiologicalFather a:hasFather )`	Having a biological father is a kind of having a father.

The object property *a:hasUncle* occurs in an object subproperty axiom involving a property chain, so it is not simple. Consequently, the object property *a:hasRelative* is not simple either, because *a:hasUncle* is a nonsimple subproperty of *a:hasRelative*. In contrast, the object property *a:hasBiologicalFather* is simple, and so is *a:hasFather*.

11.2 The Restrictions on the Axiom Closure

The axioms *Ax* satisfy the *global restrictions* of OWL 2 if the following six conditions hold:

- Each class expression in *Ax* of a type from the following list contains an object property expression that is simple in *Ax*:
    - **MinObjectCardinality**, **MaxObjectCardinality**, **ExactObjectCardinality**, and **ObjectHasSelf** .
- Each axiom in *Ax* of a type from the following list contains an object property expression that is simple in *Ax*:
    - **FunctionalObjectProperty**, **InverseFunctionalObjectProperty**, **IrreflexiveObjectProperty**, **AsymmetricObjectProperty**, and **DisjointObjectProperties**.
- The *owl:topDataProperty* property occurs in *Ax* only in the **superDataPropertyExpression** part of **SubDataPropertyOf** axioms.
- A strict partial order (i.e., an irreflexive and transitive relation) < on *AllOPE(Ax)* exists that fulfills the following conditions:
    - $OP_1 < OP_2$ if and only if $INV(OP_1) < OP_2$ for all object properties OP_1 and OP_2 occurring in *AllOPE(Ax)*.
    - If $OPE_1 < OPE_2$ holds, then $OPE_2 \rightarrow^* OPE_1$ does not hold;
    - Each axiom in *Ax* of the form `SubPropertyOf( PropertyChain( OPE1 ... OPEn ) OPE )` with n ≥ 2 fulfills the following conditions:
        - `OPE` is equal to *owl:topObjectProperty*, or
        - n = 2 and $OPE_1 = OPE_2 = OPE$, or
        - $OPE_i < OPE$ for each 1 ≤ i ≤ n, or
        - $OPE_1 = OPE$ and $OPE_i < OPE$ for each 2 ≤ i ≤ n, or
        - $OPE_n = OPE$ and $OPE_i < OPE$ for each 1 ≤ i ≤ n-1.
- No axiom in *Ax* of the following form contains anonymous individuals:
    - **SameIndividual**, **DifferentIndividuals**, **NegativeObjectPropertyAssertion**, and **NegativeDataPropertyAssertion**.
- A forest *F* over the anonymous individuals in *Ax* exists such that the following conditions are satisfied, for `OPE` an object property expression, `_:x` and `_:y` anonymous individuals, and `a` a named individual:
    - for each assertion in *Ax* of the form `PropertyAssertion( OPE _:x _:y )`, either `_:x` is a child of `_:y` or `_:y` is a child of `_:x` in *F*;
    - for each pair of anonymous individuals `_:x` and `_:y` such that `_:y` is a child of `_:x` in *F*, the axiom closure *Ax* contains at most one assertion of the form `PropertyAssertion( OPE _:x _:y )` or `PropertyAssertion( OPE _:y _:x )`; and
    - for each anonymous individual `_:x` that is a root in *F*, the axiom closure *Ax* contains at most one assertion of the form `PropertyAssertion( OPE _:x a )` or `PropertyAssertion( OPE a _:x )`.

Example:

The first two restrictions merely prohibit the usage of nonsimple properties in number restrictions and in certain axioms about object properties. The third restriction limits the usage of *owl:topDataProperty*. Without it, *owl:topDataProperty* could be used to write axioms about datatypes, which would invalidate Theorem DS1 from the OWL 2 Direct Semantics [OWL 2 Direct Semantics].

Example:

The main goal of the fourth restriction is to prevent cyclic definitions involving object subproperty axioms with property chains. Consider the following ontology:

`SubPropertyOf( PropertyChain( a:hasFather a:hasBrother ) a:hasUncle )`	The brother of someone's father is that person's uncle.
`SubPropertyOf( PropertyChain( a:hasChild a:hasUncle ) a:hasBrother )`	The uncle of someone's child is that person's brother.

The first axiom defines *a:hasUncle* in terms of *a:hasBrother*, while the second axiom defines *a:hasBrother* in terms of *a:hasUncle*. These two axioms are thus cyclic: the first one depends on the second one and vice versa. Such cyclic definitions are known to lead to undecidability of the basic reasoning problems. Thus, these two axioms mentioned above cannot occur together in an axiom closure of an OWL 2; however, each axiom alone may be allowed (depending on the other axioms in the closure).

Example:

A particular kind of cyclic definitions is known not to lead to decidability problems. Consider the following ontology:

`SubPropertyOf( PropertyChain( a:hasChild a:hasSibling ) a:hasChild )`	The sibling of someone's child is that person's child.

The above definition is cyclic, since the object property *a:hasChild* occurs in both the subproperty chain and as a superproperty. Axioms of this form, however, do not violate the global restrictions of OWL 2.

Example:

The fifth and the sixth restriction ensure that each OWL 2 ontology with anonymous individuals can be transformed to an equivalent ontology without anonymous individuals. Roughly speaking, this is possible if property assertions connect anonymous individuals in a tree-like way. Consider the following ontology:

`PropertyAssertion( a:hasChild a:Francis _:x )`	Francis has some (unknown) child.
`PropertyAssertion( a:hasChild _:x a:Meg )`	This unknown child has Meg...
`PropertyAssertion( a:hasChild _:x a:Chris )`	...Chris...
`PropertyAssertion( a:hasChild _:x a:Stewie )`	...and Stewie as children.

The connections between individuals *a:Francis*, *a:Meg*, *a:Chris*, and *a:Stewie* can be understood as a tree that contains `_:x` as its internal node. Because of that, the anonymous individuals can be "rolled-up"; that is, these four assertions can be replaced by the following equivalent assertion:

```
ClassAssertion(
    SomeValuesFrom( a:hasChild
        IntersectionOf(
            HasValue( a:hasChild a:Meg )
            HasValue( a:hasChild a:Chris )
            HasValue( a:hasChild a:Stewie )
        )
    )
    a:Francis
)
```

If the anonymous individuals were allowed to be connected by properties in arbitrary ways (and, in particular, in cycles), such a transformation would clearly be impossible. This transformation, however, is necessary in order to reduce the basic inference problems in OWL 2 to the appropriate description logic reasoning problems with known computational properties [SROIQ].

12 Appendix: Internet Media Type, File Extension, and Macintosh File Type

Contact
: Ivan Herman / Sandro Hawke

See also
: How to Register a Media Type for a W3C Specification Internet Media Type registration, consistency of use TAG Finding 3 June 2002 (Revised 4 September 2002)

The Internet Media Type / MIME Type for the OWL functional-style Syntax is `text/owl-functional`.

It is recommended that OWL functional-style Syntax files have the extension `.ofn` (all lowercase) on all platforms.

It is recommended that OWL functional-style Syntax files stored on Macintosh HFS file systems be given a file type of `TEXT`.

The information that follows will be submitted to the IESG for review, approval, and registration with IANA.

Type name
: text

Subtype name
: owl-functional

Required parameters
: None

Optional parameters
: charset This parameter may be required when transfering non-ASCII data across some protocols. If present, the value of charset should be UTF-8.

Encoding considerations
: The syntax of the OWL functional-style Syntax is expressed over code points in Unicode [*UNICODE*]. The encoding should be UTF-8 [*RFC3629*], but other encodings are allowed.

Security considerations
: The OWL functional-style Syntax uses IRIs as term identifiers. Applications interpreting data expressed in the OWL functional-style Syntax should address the security issues of Internationalized Resource Identifiers (IRIs) [*RFC3987*] Section 8, as well as Uniform Resource Identifiers (URI): Generic Syntax [*RFC3986*] Section 7. Multiple IRIs may have the same appearance. Characters in different scripts may look similar (a Cyrillic "o" may appear similar to a Latin "o"). A character followed by combining characters may have the same visual representation as another character (LATIN SMALL LETTER E followed by COMBINING ACUTE ACCENT has the same visual representation as LATIN SMALL LETTER E WITH ACUTE). Any person or application that is writing or interpreting data in the OWL functional-style Syntax must take care to use the IRI that matches the intended semantics, and avoid IRIs that may look similar. Further information about matching of similar characters can be found in Unicode Security Considerations [*UNISEC*] and Internationalized Resource Identifiers (IRIs) [*RFC3987*] Section 8.

Interoperability considerations
: There are no known interoperability issues.

Published specification
: This specification.

Applications which use this media type
: No widely deployed applications are known to currently use this media type. It is expected that OWL tools will use this media type in the future.

Additional information

None.

Magic number(s)
: OWL functional-style Syntax documents may have the strings 'Namespace:' or 'Ontology:' (case dependent) near the beginning of the document.

File extension(s)
: ".ofn"

Base IRI
: There are no constructs in the OWL functional-style Syntax to change the Base IRI.

Macintosh file type code(s)
: "TEXT"

Person & email address to contact for further information
: Ivan Herman <ivan@w3.org> / Sandro Hawke <sandro@w3.org>

Intended usage
: COMMON

Restrictions on usage
: None

Author/Change controller
: The OWL functional-style Syntax is the product of the W3C OWL Working Group; W3C reserves change control over this specification.

13 Complete Grammar (Normative)

This section contains the complete grammar of the functional-style syntax defined in this specification document. The grammar has been split into two parts. The following productions define the basic concepts such as IRIs and ontologies.

full-IRI := '*IRI as defined in [RFC3987], enclosed in a pair of < (U+3C) and > (U+3E) characters*'
NCName := '*as defined in [XML Namespaces]*'
irelative-ref := '*as defined in [RFC3987]*'
nonNegativeInteger := '*a nonempty finite sequence of digits between 0 and 9*'
quotedString := '*a finite sequence of characters in which " (U+22) and \ (U+5C) occur only in pairs of the form \" (U+22, U+5C) and \\ (U+22, U+22), enclosed in a pair of " (U+22) characters*'
languageTag := '*a nonempty (not quoted) string defined as specified in BCP 47 [BCP 47]*'
nodeID := '*a node ID of the form _:name as specified in the N-Triples specification [RDF Test Cases]*'

namespace := **full-IRI**
prefix := **NCName**
reference := **irelative-ref**
curie := [[**prefix**] ':'] **reference**

```
IRI := full-IRI | curie


ontologyDocument := { prefixDefinition } Ontology
prefixDefinition := 'Namespace' '(' [ prefix ] '=' namespace ')'
Ontology :=
    'Ontology' '(' [ ontologyIRI [ versionIRI ] ]
        directlyImportsDocuments
        ontologyAnnotations
        axioms
    ')'
ontologyIRI := IRI
versionIRI := IRI
directlyImportsDocuments := { 'Import' '(' IRI ')' }
ontologyAnnotations := { Annotation }
axioms := { Axiom }


Declaration := 'Declaration' '(' axiomAnnotations Entity ')'
Entity :=
    'Class' '(' Class ')' |
    'Datatype' '(' Datatype ')' |
    'ObjectProperty' '(' ObjectProperty ')' |
    'DataProperty' '(' DataProperty ')' |
    'AnnotationProperty' '(' AnnotationProperty ')' |
    'NamedIndividual' '(' NamedIndividual ')'


AnnotationSubject := IRI | AnonymousIndividual
AnnotationValue := AnonymousIndividual | IRI | Literal
axiomAnnotations := { Annotation }

Annotation := 'Annotation' '(' annotationAnnotations
AnnotationProperty AnnotationValue ')'
annotationAnnotations  := { Annotation }

AnnotationAxiom := AnnotationAssertion | SubAnnotationPropertyOf |
AnnotationPropertyDomain | AnnotationPropertyRange

AnnotationAssertion := 'AnnotationAssertion' '(' axiomAnnotations
AnnotationProperty AnnotationSubject AnnotationValue ')'

SubAnnotationPropertyOf := 'SubPropertyOf' '(' axiomAnnotations
subAnnotationProperty superAnnotationProperty ')'
subAnnotationProperty := AnnotationProperty
superAnnotationProperty := AnnotationProperty

AnnotationPropertyDomain := 'PropertyDomain' '(' axiomAnnotations
AnnotationProperty IRI ')'

AnnotationPropertyRange := 'PropertyRange' '(' axiomAnnotations
AnnotationProperty IRI ')'
```

The following productions define various constructors of OWL 2.

```
Class := IRI

Datatype := IRI

ObjectProperty := IRI

DataProperty := IRI

AnnotationProperty := IRI

Individual := NamedIndividual | AnonymousIndividual

NamedIndividual := IRI

AnonymousIndividual := nodeID

Literal := typedLiteral | abbreviatedXSDStringLiteral |
abbreviatedRDFTextLiteral
typedLiteral := lexicalValue '^^' Datatype
lexicalValue := quotedString
abbreviatedXSDStringLiteral := quotedString
abbreviatedRDFTextLiteral := quotedString '@' languageTag


ObjectPropertyExpression := ObjectProperty | InverseObjectProperty

InverseObjectProperty := 'InverseOf' '(' ObjectProperty ')'

DataPropertyExpression := DataProperty
```

```
DataRange  :=
      Datatype  |
      DataIntersectionOf  |
      DataUnionOf  |
      DataComplementOf  |
      DataOneOf  |
      DatatypeRestriction

DataIntersectionOf  := 'IntersectionOf' '(' DataRange DataRange {
DataRange } ')'

DataUnionOf  := 'UnionOf' '(' DataRange DataRange { DataRange }
')'

DataComplementOf  := 'ComplementOf' '(' DataRange ')'

DataOneOf  := 'OneOf' '(' Literal { Literal } ')'

DatatypeRestriction  := 'DatatypeRestriction' '(' Datatype
constrainingFacet restrictionValue { constrainingFacet restrictionValue }
')'
constrainingFacet  := IRI
restrictionValue  := Literal



ClassExpression  :=
      Class  |
      ObjectIntersectionOf  |  ObjectUnionOf  |  ObjectComplementOf  |
ObjectOneOf  |
      ObjectSomeValuesFrom  |  ObjectAllValuesFrom  |  ObjectHasValue  |
ObjectHasSelf  |
      ObjectMinCardinality  |  ObjectMaxCardinality  |  ObjectExactCardinality
|
      DataSomeValuesFrom  |  DataAllValuesFrom  |  DataHasValue  |
      DataMinCardinality  |  DataMaxCardinality  |  DataExactCardinality

ObjectIntersectionOf  := 'IntersectionOf' '(' ClassExpression
ClassExpression { ClassExpression } ')'

ObjectUnionOf  := 'UnionOf' '(' ClassExpression ClassExpression {
ClassExpression } ')'

ObjectComplementOf  := 'ComplementOf' '(' ClassExpression ')'
```

ObjectOneOf := 'OneOf' '(' **Individual** { **Individual** }')'

ObjectSomeValuesFrom := 'SomeValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

ObjectAllValuesFrom := 'AllValuesFrom' '(' **ObjectPropertyExpression** **ClassExpression** ')'

ObjectHasValue := 'HasValue' '(' **ObjectPropertyExpression** **Individual** ')'

ObjectHasSelf := 'HasSelf' '(' **ObjectPropertyExpression** ')'

ObjectMinCardinality := 'MinCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

ObjectMaxCardinality := 'MaxCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

ObjectExactCardinality := 'ExactCardinality' '(' **nonNegativeInteger** **ObjectPropertyExpression** [**ClassExpression**] ')'

DataSomeValuesFrom := 'SomeValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

DataAllValuesFrom := 'AllValuesFrom' '(' **DataPropertyExpression** { **DataPropertyExpression** } **DataRange** ')'

DataHasValue := 'HasValue' '(' **DataPropertyExpression** **Literal** ')'

DataMinCardinality := 'MinCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

DataMaxCardinality := 'MaxCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

DataExactCardinality := 'ExactCardinality' '(' **nonNegativeInteger** **DataPropertyExpression** [**DataRange**] ')'

Axiom := **Declaration** | **ClassAxiom** | **ObjectPropertyAxiom** | **DataPropertyAxiom** | **HasKey** | **Assertion** | **AnnotationAxiom**

```
ClassAxiom := SubClassOf | EquivalentClasses | DisjointClasses |
DisjointUnion

SubClassOf := 'SubClassOf' '(' axiomAnnotations
subClassExpression superClassExpression ')'
subClassExpression := ClassExpression
superClassExpression := ClassExpression

EquivalentClasses := 'EquivalentClasses' '(' axiomAnnotations
ClassExpression ClassExpression { ClassExpression } ')'

DisjointClasses := 'DisjointClasses' '(' axiomAnnotations
ClassExpression ClassExpression { ClassExpression } ')'

DisjointUnion := 'DisjointUnion' '(' axiomAnnotations Class
disjointClassExpressions ')'
disjointClassExpressions := ClassExpression ClassExpression {
ClassExpression }



ObjectPropertyAxiom :=
    SubObjectPropertyOf | EquivalentObjectProperties |
    DisjointObjectProperties | InverseObjectProperties |
    ObjectPropertyDomain | ObjectPropertyRange |
    FunctionalObjectProperty | InverseFunctionalObjectProperty |
    ReflexiveObjectProperty | IrreflexiveObjectProperty |
    SymmetricObjectProperty | AsymmetricObjectProperty |
    TransitiveObjectProperty

SubObjectPropertyOf := 'SubPropertyOf' '(' axiomAnnotations
subObjectPropertyExpressions superObjectPropertyExpression ')'
subObjectPropertyExpressions := ObjectPropertyExpression |
propertyExpressionChain
propertyExpressionChain := 'PropertyChain' '('
ObjectPropertyExpression ObjectPropertyExpression {
ObjectPropertyExpression } ')'
superObjectPropertyExpression := ObjectPropertyExpression

EquivalentObjectProperties := 'EquivalentProperties' '('
axiomAnnotations ObjectPropertyExpression ObjectPropertyExpression {
ObjectPropertyExpression } ')'

DisjointObjectProperties := 'DisjointProperties' '('
axiomAnnotations ObjectPropertyExpression ObjectPropertyExpression {
```

ObjectPropertyExpression } ')'

ObjectPropertyDomain := 'PropertyDomain' '(' **axiomAnnotations** **ObjectPropertyExpression** **ClassExpression** ')'

ObjectPropertyRange := 'PropertyRange' '(' **axiomAnnotations** **ObjectPropertyExpression** **ClassExpression** ')'

InverseObjectProperties := 'InverseProperties' '(' **axiomAnnotations** **ObjectPropertyExpression** **ObjectPropertyExpression** ')'

FunctionalObjectProperty := 'FunctionalProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

InverseFunctionalObjectProperty := 'InverseFunctionalProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

ReflexiveObjectProperty := 'ReflexiveProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

IrreflexiveObjectProperty := 'IrreflexiveProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

SymmetricObjectProperty := 'SymmetricProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

AsymmetricObjectProperty := 'AsymmetricProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

TransitiveObjectProperty := 'TransitiveProperty' '(' **axiomAnnotations** **ObjectPropertyExpression** ')'

DataPropertyAxiom :=
SubDataPropertyOf | **EquivalentDataProperties** | **DisjointDataProperties** |
DataPropertyDomain | **DataPropertyRange** | **FunctionalDataProperty**

SubDataPropertyOf := 'SubPropertyOf' '(' **axiomAnnotations** **subDataPropertyExpression** **superDataPropertyExpression** ')'
subDataPropertyExpression := **DataPropertyExpression**
superDataPropertyExpression := **DataPropertyExpression**

EquivalentDataProperties := 'EquivalentProperties' '('

axiomAnnotations **DataPropertyExpression** **DataPropertyExpression** { **DataPropertyExpression** } ')'

DisjointDataProperties := 'DisjointProperties' '(' **axiomAnnotations** **DataPropertyExpression** **DataPropertyExpression** { **DataPropertyExpression** } ')'

DataPropertyDomain := 'PropertyDomain' '(' **axiomAnnotations** **DataPropertyExpression** **ClassExpression** ')'

DataPropertyRange := 'PropertyRange' '(' **axiomAnnotations** **DataPropertyExpression** **DataRange** ')'

FunctionalDataProperty := 'FunctionalProperty' '(' **axiomAnnotations** **DataPropertyExpression** ')'

HasKey := 'HasKey' '(' **axiomAnnotations** **ClassExpression** **ObjectPropertyExpression** | **DataPropertyExpression** { **ObjectPropertyExpression** | **DataPropertyExpression** } ')'

Assertion :=
    SameIndividual | **DifferentIndividuals** | **ClassAssertion** |
    ObjectPropertyAssertion | **NegativeObjectPropertyAssertion** |
    DataPropertyAssertion | **NegativeDataPropertyAssertion**

sourceIndividual := **Individual**
targetIndividual := **Individual**
targetValue := **Literal**

SameIndividual := 'SameIndividual' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

DifferentIndividuals := 'DifferentIndividuals' '(' **axiomAnnotations** **Individual** **Individual** { **Individual** } ')'

ClassAssertion := 'ClassAssertion' '(' **axiomAnnotations** **ClassExpression** **Individual** ')'

ObjectPropertyAssertion := 'PropertyAssertion' '(' **axiomAnnotations** **ObjectPropertyExpression** **sourceIndividual** **targetIndividual** ')'

NegativeObjectPropertyAssertion := `'NegativePropertyAssertion'` `'('` **axiomAnnotations** **objectPropertyExpression** **sourceIndividual** **targetIndividual** `')'`

DataPropertyAssertion := `'PropertyAssertion'` `'('` **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** `')'`

NegativeDataPropertyAssertion := `'NegativePropertyAssertion'` `'('` **axiomAnnotations** **DataPropertyExpression** **sourceIndividual** **targetValue** `')'`

14 Index

Editor's Note: The index will be created for the final version of the document.

15 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 is the product of the OWL Working Group (see below) whose members deserve recognition for their time and commitment. The editors extend special thanks to Bijan Parsia (University of Manchester), Mike Smith (Clark & Parsia) and Vojtech Svatek (K-Space) for their thorough reviews.

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), 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), Boris Motik (Oxford University), Jeff Pan (University of Aberdeen), Bijan Parsia (University of Manchester), Peter F. Patel-Schneider (Bell Labs Research, Alcatel-Lucent), Alan Ruttenberg (Science Commons), Uli Sattler (University of Manchester), Michael Schneider (FZI), Mike Smith (Clark & Parsia), Evan Wallace (NIST), and Zhe Wu (Oracle Corporation). We would also like to thank past members of the working group: Jeremy Carroll, Jim Hendler and Vipul Kashyap.

16 References

[OWL 2 Direct Semantics]
OWL 2 Web Ontology Language:Direct Semantics Boris Motik, Peter F. Patel-Schneider, Bernardo Cuenca Grau, eds. W3C Editor's Draft, 02 December 2008, http://www.w3.org/2007/OWL/draft/ED-owl2-semantics-20081202/. Latest version available at http://www.w3.org/2007/OWL/draft/owl2-semantics/.

[OWL 2 XML Syntax]
OWL 2 Web Ontology Language:XML Serialization Boris Motik, Peter Patel-Schneider, eds. W3C Editor's Draft, 02 December 2008, http://www.w3.org/2007/OWL/draft/ED-owl2-xml-serialization-20081202/. Latest version available at http://www.w3.org/2007/OWL/draft/owl2-xml-serialization/.

[SROIQ]
The Even More Irresistible SROIQ. Ian Horrocks, Oliver Kutz and Uli Sattler. In Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). AAAI Press, 2006.

[XML]
Extensible Markup Language (XML) 1.1. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau and John Cowan, eds. W3C Recommendation 16 August 2006, edited in place 29 September 2006.

[XML Namespaces]
Namespaces in XML 1.0 (Second Edition). Tim Bray, Dave Hollander, Andrew Layman and Richard Tobin, eds. W3C Recommendation 16 August 2006.

[XML Schema Datatypes]
W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, H. S. Thompson, eds. W3C Working Draft 20 June 2008.

[RDF Syntax]
RDF/XML Syntax Specification (Revised). Dave Beckett, Editor, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/rdf-syntax-grammar/.

[BCP 47]
BCP 47 - Tags for Identifying Languages. A. Phillips, M. Davis, eds., IETF, September 2006, http://www.rfc-editor.org/rfc/bcp/bcp47.txt.

[RFC 2119]
RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. Network Working Group, S. Bradner. Internet Best Current Practice, March 1997.

[RFC3629]
UTF-8, a transformation format of ISO 10646, F. Yergeau, November 2003, http://www.ietf.org/rfc/rfc3629.txt

[RFC3986]
RFC 3986 *Uniform Resource Identifier (URI): Generic Syntax*, T. Berners-Lee, R. Fielding and L. Masinter, January 2005, http://www.ietf.org/rfc/rfc3986.txt

[RFC3987]
RFC 3987 - Internationalized Resource Identifiers (IRIs). M. Duerst and M. Suignard. IETF, January 2005, http://www.ietf.org/rfc/rfc3987.txt.

[OWL Semantics and Abstract Syntax]
OWL Web Ontology Language: Semantics and Abstract Syntax Peter F. Patel-Schneider, Patrick Hayes and Ian Horrocks, eds. W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-owl-semantics-20040210/. Latest version available at http://www.w3.org/TR/owl-semantics/.

[CURIE]
CURIE Syntax 1.0: A syntax for expressing Compact URIs. M. Birbeck and S. McCarron, Editors, W3C Working Draft, 26 November 2007, http://www.w3.org/TR/2007/WD-curie-20071126/.

[RDF Test Cases]
RDF Test Cases. Jan Grant and Dave Beckett, Editors, W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-testcases/.

[ISO/IEC 10646]
ISO/IEC 10646-1:2000. Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane and ISO/IEC 10646-2:2001. Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 2: Supplementary Planes, as, from time to time, amended, replaced by a new edition or expanded by the addition of new parts. [Geneva]: International Organization for Standardization. ISO (International Organization for Standardization).

[DL-Safe]
Query Answering for OWL-DL with Rules. Boris Motik, Ulrike Sattler and Rudi Studer. Journal of Web Semantics: Science, Services and Agents on the World Wide Web, 3(1):41–60, 2005.

[IEEE 754]
IEEE Standard for Binary Floating-Point Arithmetic. Standards Committee of the IEEE Computer Society

[ISO 8601:2004]
ISO 8601:2004. Representations of dates and times. ISO (International Organization for Standardization).

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

[RDF:TEXT]
OWL 2 Web Ontology Language:rdf:text: A Datatype for Internationalized Text Jie Bao, Axel Polleres, Boris Motik. W3C Editor's Draft, 02 December 2008, http://www.w3.org/2007/OWL/draft/ED-rdf-text-20081202/. Latest version available at http://www.w3.org/2007/OWL/draft/rdf-text/.

[UNICODE]
The Unicode Standard Version 3.0, Addison Wesley, Reading MA, 2000, ISBN: 0-201-61633-5, http://www.unicode.org/unicode/standard/standard.html

[UNISEC]
Unicode Security Considerations, Mark Davis and Michel Suignard, July 2008, http://www.unicode.org/reports/tr36/

[MOF]
Meta Object Facility (MOF) Core Specification, version 2.0. Object Management Group, OMG Available Specification January 2006.

[UML]
OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Object Management Group, OMG Available Specification November 2007.