The WonderWeb Ontology Language Layer

Report and Tutorial

Sean Bechhofer & Ian Horrocks
University of Manchester
Kilburn Building
Oxford Road
Manchester M13 9PL
email: seanb@cs.man.ac.uk

Identifier	Del 1
Class	Deliverable
Version	1.1
Date	07-02-2003
Status	Final
Distribution	Public
Lead Partner	VUM

WonderWeb Project

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project co-ordinator:

Ian Horrocks
The Victoria University of Manchester
Department of Computer Science
Kilburn Building
Oxford Road
Manchester M13 9PL
Tel: +44 161 275 6154
Fax: +44 161 275 6236
Email: wonderweb-info@lists.man.ac.uk

Contents

1 Document Structure

In this document we describe the ontology language OWL that will form part of the WonderWeb layered language architecture. OWL is being developed by a W3C working group, which includes several members of the WonderWeb project team. As the design of OWL has yet to be finalised, we also describe in detail the DAML+OIL language on which OWL is based.

The remainder of the document is structured as follows. In Section 2 we introduce the Semantic Web and the place of ontologies in the rôle played by ontologies; in Section 3 we provide a short historical overview of the developments leading up to the OWL standardisation process; in Section 4 we describe the DAML+OIL language in detail, including syntax, semantics and design rationale; in Section 5 we describe the OWL language, in particular highlighting the differences with respect to DAML+OIL; in Section 6 we provide a tutorial introduction to the OWL language; and in Section 7 we provide an example ontology illustrating the key features of the language.

Readers whose main objective is to understand the key features of the language, or those readers who are new to (Web) ontology languages, are recommended to read Sections 2 and 3, and then to skip to Sections 6 and 7, coming back later to the intervening technical sections if required.

2 Background

The World Wide Web has been made possible through a set of widely established standards which guarantee interoperability at various levels: the TCP/IP protocol has ensured that nobody has to worry about transporting bits over the wire anymore; similarly, HTTP and HTML have provided a standard way of retrieving and presenting hyperlinked text documents. Applications were able to use this common infrastructure and this has led to the WWW as we know it now.

The current Web can be characterised as the second generation Web: the first generation Web was characterised by handwritten HTML pages; the second generation made the step to machine generated and often active HTML pages. These generations of the Web were meant for direct human processing (reading, browsing, form-filling, etc.). The third generation Web aims to make Web resources more readily accessible to automated processes by adding meta-data annotations that describe their content—this coincides with the vision that Tim Berners-Lee calls the Semantic Web in his recent book “Weaving the Web” [3].

2.1 Ontologies

If meta-data annotations are to make resources more accessible to automated agents, it is essential that their meaning can be understood by such agents. Ontologies will play a pivotal role here by providing a source of shared and precisely defined terms that can be used in such meta-data. An ontology typically consists of a hierarchical description of important concepts in a domain, along with descriptions of the properties of each concept. The

degree of formality employed in capturing these descriptions can be quite variable, ranging from natural language to logical formalisms, but increased formality and regularity clearly facilitates machine understanding.

Examples of the use of ontologies could include:

- in e-commerce sites [18], where ontologies can facilitate machine-based communication between buyer and seller, enable vertical integration of markets (see, e.g., `http://www.verticalnet.com/`), and allow descriptions to be reused in different marketplaces;
- in search engines [19], where ontologies can help searching to go beyond the current keyword-based approach, and allow pages to be found that contain syntactically different, but semantically similar words/phrases (see, e.g., `http://www.hotbot.com/`);
- in Web services [20], where ontologies can provide semantically richer service descriptions that can be more flexibly interpreted by intelligent agents.

3 Web Ontology Languages

The recognition of the key role that ontologies are likely to play in the future of the Web has led to the extension of Web markup languages in order to facilitate content description and the development of Web based ontologies, e.g., XML Schema,[1] RDF[2] (Resource Description Framework), and RDF Schema [5]. RDF Schema (RDFS) in particular is recognisable as an ontology/knowledge representation language: it talks about classes and properties (binary relations), range and domain constraints (on properties), and subclass and subproperty (subsumption) relations.

RDFS is, however, a very primitive language (the above is an almost complete description of its functionality), and more expressive power would clearly be necessary/desirable in order to describe resources in sufficient detail. Moreover, such descriptions should be amenable to *automated reasoning* if they are to be used effectively by automated processes, e.g., to determine the semantic relationship between syntactically different terms.

3.1 DAML and OIL

In 1999 the DARPA Agent Markup Language (DAML) program[3] was initiated with the aim of providing the foundations of a next generation "semantic" Web [11]. As a first step, it was decided that the adoption of a common ontology language would facilitate semantic interoperability across the various projects making up the program. RDFS was seen as a good starting point, and was already a proposed World Wide Web Consortium (W3C) standard, but it was not expressive enough to meet DAML's requirements. A new language called DAML-ONT was therefore developed that extended RDF with language

[1] `http://www.w3.org/XML/Schema/`
[2] `http://www.w3c.org/RDF/`
[3] `http://www.daml.org/`

constructors from object-oriented and frame-based knowledge representation languages. Like RDFS, DAML-ONT suffered from a rather weak semantic specification, and it was soon realised that this could lead to disagreements, both amongst humans and machines, as to the precise meaning of terms in a DAML-ONT ontology.

At around the same time, a group of (largely European) researchers with aims similar to those of the DAML researchers (i.e., to provide a foundation for the next generation Web) had designed another Web oriented ontology language called OIL (the Ontology Inference Layer) [6, 7].[4] Like DAML-ONT, OIL had an RDFS based syntax (as well as an alternative XML syntax) and a set of language constructors based on frame-based languages. The developers of OIL, however, placed a stronger emphasis on formal rigor, and the language was explicitly designed so that its semantics could be specified via a mapping to a very expressive description logic, $SHIQ$ [16].

It became obvious to both groups that their objectives could best be served by combining their efforts, the result being the merging of DAML-ONT and OIL to produce DAML+OIL. The merged language has a formal (model theoretic) semantics that provides machine and human understandability (as well as an axiomatization [8]), and a reconciliation of the language constructors from the two languages.

3.2 DAML+OIL and OWL

Until recently, the development of DAML+OIL has been undertaken by a committee largely made up of members of the two language design teams (the Joint EU/US Committee on Agent Markup Languages). More recently, DAML+OIL has been submitted to W3C[5] as a technical note and now forms the basis for the W3C's proposed Web ontology language (OWL) which the Web-Ontology Working Group has been mandated to deliver.[6]

As development of OWL is still underway, we will first discuss DAML+OIL and then describe how OWL (or at least the current proposal for OWL) differs from DAML+OIL.

4 DAML+OIL

DAML+OIL describes the structure of a domain in terms of *classes* and *properties*. A DAML+OIL ontology consists of a set of *axioms* that constrain the meaning of terms, e.g., subsumption relationships between classes or properties. Instances of classes (properties) are assumed to be RDF resources[7] (pairs of RDF resources). Asserting that a given resource (pair of resources) is an instance of a given DAML+OIL class (property) is left to RDF, a task for which it is well suited.

[4] Work on OIL was supported by the IST OntoKnowledge project (see `http://www.ontoknowledge.org/` and `http://www.ontoknowledge.org/oil/`).

[5] `http://www.w3.org/Submission/2001/12/`

[6] `http://www.w3c.org/2001/sw/WebOnt/`

[7] Everything describable by RDF is called a resource. A resource could be web accessible, e.g., a web page or part of a web page, but it could also be an object that is not directly accessible via the web, e.g., a person. Resources are named by URIs plus optional anchor ids. See `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/` for more details.

Constructor	DL Syntax	Example
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$	Human $\sqcap$ Male
unionOf	$C_1 \sqcup \ldots \sqcup C_n$	Doctor $\sqcup$ Lawyer
complementOf	$\neg C$	$\neg$Male
oneOf	$\{x_1 \ldots x_n\}$	$\{$john, mary$\}$
toClass	$\forall P.C$	$\forall$hasChild.Doctor
hasClass	$\exists P.C$	$\exists$hasChild.Lawyer
hasValue	$\exists P.\{x\}$	$\exists$citizenOf.$\{$USA$\}$
minCardinalityQ	$\geqslant nP.C$	$\geqslant$2hasChild.Lawyer
maxCardinalityQ	$\leqslant nP.C$	$\leqslant$1hasChild.Male
cardinalityQ	$=nP.C$	$=$1 hasParent.Female

Figure 1: DAML+OIL class constructors

4.1 Language Constructors

From a formal point of view, DAML+OIL can be seen to be equivalent to a very expressive description logic, with a DAML+OIL ontology corresponding to the Tbox, and RDF type and property assertions corresponding to the Abox. As in a DL, DAML+OIL classes can be names (URIs) or *expressions*, and a variety of *constructors* are provided for building class expressions, with the expressive power of the language being determined by the class (and property) constructors supported, and by the kinds of axiom supported.

Figure 1 summarises the constructors supported by DAML+OIL, where C (possibly subscripted) is a class, P is a property, x (possibly subscripted) is an individual and n is a non-negative integer. The standard DL syntax is used for compactness, as the RDF syntax is rather verbose. In the RDF syntax, for example, Human $\sqcap$ Male would be written as

```
<daml:Class>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Human"/>
    <daml:Class rdf:about="#Male"/>
  </daml:intersectionOf>
</daml:Class>
```

while $\geqslant$2hasChild.Lawyer would be written as

```
<daml:Restriction daml:minCardinalityQ="2">
  <daml:onProperty rdf:resource="#hasChild"/>
  <daml:hasClassQ rdf:resource="#Lawyer"/>
</daml:Restriction>
```

4.2 Datatypes

DAML+OIL supports the full range of datatypes in XML Schema: the so called primitive datatypes such as string, decimal or float, as well as more complex derived datatypes

such as integer sub-ranges. This is facilitated by maintaining a clean separation between instances of "object" classes (defined using the ontology language) and instances of datatypes (defined using the XML Schema type system). In particular, the domain of interpretation of object classes is disjoint from the domain of interpretation of datatypes, so that an instance of an object class (e.g., the individual "Italy") can never have the same denotation as a value of a datatype (e.g., the integer 5), and that the set of object properties (which map individuals to individuals) is disjoint from the set of datatype properties (which map individuals to datatype values).

The disjointness of object and datatype domains was motivated by both philosophical and pragmatic considerations:

- Datatypes are considered to be already sufficiently structured by the built-in predicates, and it is, therefore, not appropriate to form new classes of datatype values using the ontology language [12].
- The simplicity and compactness of the ontology language are not compromised: even enumerating all the XML Schema datatypes would add greatly to its complexity, while adding a logical theory for each datatype, even if it were possible, would lead to a language of monumental proportions.
- The semantic integrity of the language is not compromised—defining theories for all the XML Schema datatypes would be difficult or impossible without extending the language in directions whose semantics would be difficult to capture within the existing framework.
- The "implementability" of the language is not compromised—a hybrid reasoner can easily be implemented by combining a reasoner for the "object" language with one capable of deciding satisfiability questions with respect to conjunctions of (possibly negated) datatypes [15].

From a theoretical point of view, this design means that the ontology language can specify constraints on data values, but as data values can never be instances of object classes they cannot apply additional constraints to elements of the object domain. This allows the type system to be extended without having any impact on the ontology language, and vice versa. Similarly, the formal properties of hybrid reasoners are determined by those of the two components; in particular, the combined reasoner will be sound and complete if both components are sound and complete.

From a practical point of view, DAML+OIL implementations can choose to support some or all of the XML Schema datatypes. For supported datatypes, they can either implement their own type checker/validator or rely on some external component. The job of a type checker/validator is simply to take zero or more data values and one or more datatypes, and determine if there exists any data value that is equal to every one of the specified data values and is an instance of every one of the specified data types.

4.3 Ontology Axioms

Figure 2 summarises the axioms supported by DAML+OIL, where C (possibly subscripted) is a class, P (possibly subscripted) is a property, P^- is the inverse of P, P^+

Axiom	DL Syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human $\sqsubseteq$ Animal $\sqcap$ Biped
sameClassAs	$C_1 \equiv C_2$	Man $\equiv$ Human $\sqcap$ Male
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter $\sqsubseteq$ hasChild
samePropertyAs	$P_1 \equiv P_2$	cost $\equiv$ price
disjointWith	$C_1 \sqsubseteq \neg C_2$	Male $\sqsubseteq \neg$Female
sameIndividualAs	$\{x_1\} \equiv \{x_2\}$	$\{$President_Bush$\} \equiv \{$G_W_Bush$\}$
differentIndividualFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	$\{$john$\} \sqsubseteq \neg\{$peter$\}$
inverseOf	$P_1 \equiv P_2^-$	hasChild $\equiv$ hasParent$^-$
transitiveProperty	$P^+ \sqsubseteq P$	ancestor$^+ \sqsubseteq$ ancestor
uniqueProperty	$\top \sqsubseteq \leqslant 1P$	$\top \sqsubseteq \leqslant$1hasMother
unambiguousProperty	$\top \sqsubseteq \leqslant 1P^-$	$\top \sqsubseteq \leqslant$1isMotherOf$^-$

Figure 2: DAML+OIL axioms

is the transitive closure of P, x (possibly subscripted) is an individual and $\top$ is an abbreviation for $A \sqcup \neg A$ for some class A. These axioms make it possible to assert subsumption or equivalence with respect to classes or properties, the disjointness of classes, the equivalence or non-equivalence of individuals, and various properties of properties.

A crucial feature of DAML+OIL is that subClassOf and sameClassAs axioms can be applied to arbitrary class expressions. This provides greatly increased expressive power with respect to standard frame-based languages where such axioms are invariably restricted to so called *definitions*, where the left hand side is an atomic name, there is only one such axiom per name, and there are no definitional cycles (the class on the right hand side of an axiom cannot refer, either directly or indirectly, to the class name on the left hand side).

A consequence of the expressive power of DAML+OIL is that all of the class and individual axioms, as well as the uniqueProperty and unambiguousProperty axioms, can be reduced to subClassOf and sameClassAs axioms (as can be seen from the DL syntax). In fact sameClassAs could also be reduced to subClassOf, as a sameClassAs axiom $C \equiv D$ is trivially equivalent to a pair of subClassOf axioms $C \sqsubseteq D$ and $D \sqsubseteq C$. Moreover, the distinction between Tbox and Abox breaks down, as Abox assertions can be expressed using Tbox axioms. E.g., an assertion that an individual x is an instance of a class C (written $x \in C$) can be expressed as $\{x\} \sqsubseteq C$, and an assertion that a tuple $\langle x, y \rangle$ is an instance of a property P (written $\langle x, y \rangle \in P$) can be expressed as $\{x\} \sqsubseteq \exists P.\{y\}$.

As far as property axioms are concerned, it is possible to assert that a given property is unique (functional), unambiguous (inverse functional) or transitive (i.e., that its interpretation must be closed under composition). It is also possible to assign a name to the inverse of a property, thus allowing inverse properties to be used in class expressions. Transitive properties are preferred over transitive closure as this has been shown to facilitate the design of (efficient) algorithms [17].

Constructor	DL Syntax	Semantics
`intersectionOf`	$C_1 \sqcap \ldots \sqcap C_n$	$C_1^I \cap \ldots \cap C_n^I$
`unionOf`	$C_1 \sqcup \ldots \sqcup C_n$	$C_1^I \cup \ldots \cup C_n^I$
`complementOf`	$\neg C$	$\Delta^I \setminus C^I$
`oneOf`	$\{x_1 \ldots x_n\}$	$\{x_1^I \ldots x_n^I\}$
`toClass`	$\forall P.C$	$\{x \mid \forall y.\langle x,y\rangle \in P^I \text{ implies } y \in C^I\}$
`hasClass`	$\exists P.C$	$\{x \mid \exists y.\langle x,y\rangle \in P^I \text{ and } y \in C^I\}$
`hasValue`	$\exists P.\{x\}$	$\{y \mid \langle y,x^I\rangle \in P^I\}$
`minCardinalityQ`	$\geqslant nP.C$	$\{x \mid \sharp\{y \mid \langle x,y\rangle \in P^I \text{ and } y \in C^I\} \geqslant n\}$
`maxCardinalityQ`	$\leqslant nP.C$	$\{x \mid \sharp\{y \mid \langle x,y\rangle \in P^I \text{ and } y \in C^I\} \leqslant n\}$
`cardinalityQ`	$=nP.C$	$\{x \mid \sharp\{y \mid \langle x,y\rangle \in P^I \text{ and } y \in C^I\} = n\}$

Figure 3: Semantics of DAML+OIL class constructors

4.4 Semantics

The meaning of the language is defined by a standard model-theoretic semantics.[8] The semantics is based on interpretations, where an interpretation consists of a domain of discourse and an interpretation function. The domain is divided into two disjoint sets, the "object domain" Δ_O^I and the "datatype domain" Δ_D^I. The interpretation function ^{I} maps classes into subsets of the object domain, individuals into elements of the object domain, datatypes into subsets of the datatype domain and data values into elements of the datatype domain. In addition, two disjoint sets of properties are distinguished: object properties and datatype properties. The interpretation function maps the former into subsets of $\Delta_O^I \times \Delta_O^I$ and the latter into subsets of $\Delta_O^I \times \Delta_D^I$.

The interpretation function is extended to concept expressions as shown in Figure 3, and an interpretation is said to satisfy a given axiom if it satisfies the conditions shown in Figure 4 (note that $\sharp\mathbf{S}$ is used to denote the cardinality of the set $\mathbf{S}$). As usual, an interpretation is called a model of an ontology O if it satisfies each of the axioms in O. An ontology O is said to be satisfiable if it has a model, and a class C is said to be satisfiable w.r.t. O if there is a model of O in which the interpretation of C is non-empty.

A full specification of the semantics of DAML+OIL can be found at `http://www.daml.org/2001/03/model-theoretic-semantics`.

4.5 Extending RDF Schema

DAML+OIL is tightly integrated with RDFS: RDFS is used to express DAML+OIL's machine readable specification,[9] and RDFS provides the only serialisation for DAML+OIL. While the dependence on RDFS has some advantages in terms of the re-use of existing RDFS infrastructure and the portability of DAML+OIL ontologies, using RDFS to completely define the structure of DAML+OIL is quite difficult as, unlike XML, RDFS is not designed for the precise specification of syntactic structure. For example, there is no way

[8] `http://www.w3.org/TR/daml+oil-model`
[9] `http://www.daml.org/2001/03/daml+oil.daml`

Axiom	DL Syntax	Semantics
subClassOf	$C_1 \sqsubseteq C_2$	$C_1^I \subseteq C_2^I$
sameClassAs	$C_1 \equiv C_2$	$C_1^I = C_2^I$
subPropertyOf	$P_1 \sqsubseteq P_2$	$P_1^I \subseteq P_2^I$
samePropertyAs	$P_1 \equiv P_2$	$P_1^I = P_2^I$
disjointWith	$C_1 \sqsubseteq \neg C_2$	$C_1^I \subseteq \neg C_2^I$
sameIndividualAs	$\{x_1\} \equiv \{x_2\}$	$x_1^I = x_2^I$
differentIndividualFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	$x_1^I \neq x_2^I$
inverseOf	$P_1 \equiv P_2^-$	$P_1^I = \{\langle x,y\rangle \mid \langle y,x\rangle \in P_2^I\}$
transitiveProperty	$P^+ \sqsubseteq P$	$(P^I)^+ \sqsubseteq P^I$
uniqueProperty	$\top \sqsubseteq \leqslant 1P$	$\Delta^I \subseteq \{x \mid \sharp\{y.\langle x,y\rangle \in P^I\} \leqslant 1\}$
unambiguousProperty	$\top \sqsubseteq \leqslant 1P^-$	$\Delta^I \subseteq \{x \mid \sharp\{y.\langle y,x\rangle \in P^I\} \leqslant 1\}$

Figure 4: Semantics of DAML+OIL axioms

in RDFS to state that a restriction (slot constraint) should consist of exactly one property (slot) and one class.

The solution to this problem adopted by DAML+OIL is to define the semantics of the language in such a way that they give a meaning to any (parts of) ontologies that conform to the RDFS specification, including "strange" constructs such as restrictions with multiple properties and classes. The meaning given to strange constructs may, however, include strange "side effects". For example, in the case of a restriction with multiple properties and classes, the semantics interpret this in the same way as a conjunction of all the restrictions that would result from taking the cross product of the specified properties and classes, but with the added (and probably unexpected) effect that all these restrictions must have the same interpretation (i.e., are equivalent).

DAML+OIL's dependence on RDFS may also have consequences for the decidability of the language. Decidability is lost when cardinality restrictions can be applied to properties that are transitive, or that have transitive sub-properties [16]. There is no way to formally capture this constraint in RDFS, so decidability in DAML+OIL depends on an informal prohibition of cardinality restrictions on non-simple properties.

4.6 Comparing DAML+OIL with OIL

It is both interesting and illuminating to compare the design of DAML+OIL with that of OIL. From the point of view of language constructs, the differences between the two are relatively trivial: although there is some difference in "keyword" vocabulary, there is usually a one to one mapping of constructors, and in the cases where the constructors are not completely equivalent, simple translations are possible.

OIL also uses RDFS for its serialisation (although it also provides a separate XML-based syntax). Consequently, OIL's RDFS based syntax would seem to be susceptible to the same difficulties as described above for DAML+OIL. However, in the case of OIL there does not seem to be an assumption that any ontology conforming to the RDFS meta-description should be a valid OIL ontology—presumably ontologies containing un-

expected usages of the meta-properties would be rejected by OIL processors as the semantics do not specify how these could be translated into $SHIQ(\mathbf{D})$. Thus, OIL and DAML+OIL take rather different positions with regard to the layering of languages on the Semantic Web.

Another effect of DAML+OIL's tight integration with RDFS is that the frame structure of OIL's syntax is much less evident: a DAML+OIL ontology is more DL-like in that it consists largely of a relatively unstructured collection of subsumption and equality axioms. This can make it more difficult to use DAML+OIL with frame based tools such as Protégé [10] or OilEd [2] because the axioms may be susceptible to many different frame-like groupings [1]. It is interesting to observe that the OWL abstract syntax has reverted to grouping axioms into frame like structures (see Section 5.1).

The treatment of individuals in OIL is also very different from that in DAML+OIL. In the first place, DAML+OIL relies wholly on RDF for assertions involving the type (class) of an individual or a relationship between a pair of objects. In the second place, DAML+OIL treats individuals occurring in the ontology (in oneOf constructs or hasValue restrictions) as true individuals (i.e., interpreted as single elements in the domain of discourse) and not as primitive concepts as is the case in OIL. This weak treatment of the oneOf construct is a well known technique for avoiding the reasoning problems that arise with existentially defined classes, and is also used, e.g., in the CLASSIC knowledge representation system [4]. Moreover, DAML+OIL makes no unique name assumption: it is possible to explicitly assert that two individuals are the same or different, or to leave their relationship unspecified.

This treatment of individuals is very powerful, and justifies intuitive inferences that would not be valid for OIL, e.g., that persons all of whose countries of residence are Italy are kinds of person that have at most one country of residence:

$$\mathsf{Person} \sqcap \forall \mathit{residence}.\{\mathsf{Italy}\} \sqsubseteq \leqslant 1\mathit{residence}$$

Unfortunately it is also very difficult to deal with computationally, and although the language is known to be decidable (it is contained in the C2 fragment of first order logic [21, 9]), there is currently no known sound and complete algorithm for ontology satisfiability [14].

5 OWL

OWL is the name given by the W3C Web Ontology (WebOnt) Working Group[10] to the ontology language standard they are developing and which is based on DAML+OIL. OWL is still under development, but the main features of the language are very similar to those of DAML+OIL. This can be seen in the three documents (working drafts) that have already been released, and which give details of the proposed language:

Feature Synopsis for OWL Lite and OWL This document gives a compact overview of the language, lists its basic syntactic elements and introduces the OWL Lite subset (`http://www.w3.org/TR/2002/WD-owl-features-20020729/`).

[10] `http://www.w3.org/2001/sw/WebOnt/`

OWL Web Ontology Language Reference This document gives a detailed specification of the RDF syntax of the language, including usage examples (`http://www.w3.org/TR/2002/WD-owl-ref-20020729/`).

OWL Web Ontology Language Abstract Syntax This document gives a high level description of the language features in an abstract syntactic form similar to that of OIL (`http://www.w3.org/TR/2002/WD-owl-absyn-20020729/`).

Currently, OWL differs from DAML+OIL in the following respects:

1. The namespace has been changed to `http://www.w3.org/2002/07/owl`.
2. Several bugs and omissions in RDF and RDF Schema have now been fixed by the RDF Core Working Group. This allows OWL to use "official" RDF syntax (in contrast to DAML+OIL). Examples include:
   - RDF now supports cyclical class and property inclusions (using rdf:subClassOf and rdf:subPropertyOf), so the relevant RDF properties can now be used in OWL.
   - RDF now supports multiple rdfs:domain and rdfs:range properties, with both being treated as equivalent to the intersection of the individual constraints.
   - RDF now includes a collection parse type (rdf:parseType="Collection") which replaces the old rdf:parseType="daml:collection".
   - RDF now has its own formal semantics (a model theory). The relationship between this and the OWL semantics is as yet unclear.
   - The RDF core working group are examining the issue of datatypes and may produce a recommendation. The issue of OWL datatypes is on hold pending the outcome of their work.
3. Some DAML+OIL properties and classes have been renamed. The following table, which shows the relevant DAML+OIL names and their corresponding OWL names, is taken from the OWL Language Reference:

DAML+OIL	OWL
`daml:hasClass`	`owl:someValuesFrom`
`daml:toClass`	`owl:allValuesFrom`
`daml:UnambiguousProperty`	`owl:InverseFunctionalProperty`
`daml:UniqueProperty`	`owl:FunctionalProperty`

4. A new type of property, owl:SymmetricProperty, has been added. As the name suggests, this denotes a symmetric property—one whose inverse is equal to itself. This could also be expressed (e.g., for the property hasSibling) using samePropertyAs and inverseOf by asserting that hasSibling is the samePropertyAs the inverseOf hasSibling:

```
<owl:ObjectProperty rdf:ID="hasSibling">
  <owl:samePropertyAs>
```

```
    <owl:ObjectProperty>
      <owl:inverseOf rdf:resource="#hasSibling"/>
    </owl:ObjectProperty>
  </owl:samePropertyAs>
</owl:ObjectProperty>
```

5. Qualified cardinality restrictions are not supported in OWL. This results in the removal of the DAML+OIL properties daml:cardinalityQ, daml:hasClassQ, daml:maxCardinalityQ and daml:minCardinalityQ. This would seem to represent a real reduction in the expressive power of OWL with respect to DAML+OIL, but it was decided by the working group that qualified cardinality restrictions are of limited practical value and are too difficult for users to understand.

5.1 Abstract Syntax

As well as the DAML+OIL style RDF syntax, the OWL specification includes an abstract syntax which provides a higher level and less cumbersome way of writing ontologies. It also has the advantage of allowing a more succinct statement of the semantics than is possible with the RDF syntax (because it is more compact, groups together semantically related statements, and allows fewer syntactic variations). A translation from this syntax to the RDF syntax is provided.

The OWL abstract syntax is very like OIL in that it provides for compound axioms resembling frames. These compound axioms use an abstract form of the standard constructors called a description.

The abstract syntax is defined using an extended BNF notation. To improve readability we will use the form of BNF where non-terminals are written in bold face (**terminal**), terminals are written in standard face (non-terminal), optional items are enclosed in square brackets ([optional]), items that may be repeated zero or more times are enclosed in braces ({repeated}), and single character terminals are enclosed in double-quotes ("(") to distinguish them from meta-symbols. Using this notation, OWL descriptions can be defined as follows:

description ::= **classID**
| **restriction**
| unionOf "(" **description** {**description**} ")"
| intersectionOf "(" **description** {**description**} ")"
| complementOf "(" **description** ")"
| oneOf "(" {**individualID**} ")"

where both classID and individualID are URI references.

Restrictions correspond to slot constraints in OIL (or in a standard frame language) and are defined as follows:

```
restriction ::= restriction "(" datavaluedPropertyID
                                {allValuesFrom=dataRange}
                                {someValuesFrom=dataRange}
                                {value=dataLiteral}
                                [cardinality] ")"
              | restriction "(" individualvaluedPropertyID
                                {allValuesFrom=description}
                                {someValuesFrom=description}
                                {value=individualID}
                                [cardinality] ")"
cardinality ::= minCardinality "(" non-negative-integer ")"
              | maxCardinality "(" non-negative-integer ")"
              | minCardinality "(" non-negative-integer ")"
                maxCardinality "(" non-negative-integer ")"
              | cardinality "(" non-negative-integer ")"
```

where datavaluedPropertyID, individualvaluedPropertyID and datatypeID are URI references, and non-negative-integer is an integer greater than or equal to 0 (zero). The two different forms of restriction reflect the treatment of datatypes in DAML+OIL, i.e., the disjointness of datatype and object domains and the disjointness of the set of datavalued properties and individualvalued properties. Restrictions on datavalued properties therefore take data ranges in the allValuesFrom and someValuesFrom fields, and data literals in the value field; restrictions on individualvalued properties take descriptions in the allValuesFrom and someValuesFrom fields, and individuals in the value field.

Data ranges and literals are defined as follows:

```
dataLiteral        ::= typedDataLiteral | untypedDataLiteral
typedDataLiteral   ::= datatypeID lexical-form
untypedDataLiteral ::= lexical-form

dataRange ::= datatypeID
            | OneOf "(" {typedDataLiteral} ")"
```

where datatypeID is a URI reference and lexical-form is just a sequence of characters (a string) giving a lexical representation of the data value. E.g., "10.0" is a valid lexical form for (amongst others) the decimal 10.0, the float 10.0E0, and the string "10.0".

Descriptions are used in compound axioms as follows:

```
axiom    ::= Class "(" classID modality {description} ")"
modality ::= complete | partial
```

where classID is a URI reference. When the modality is complete, the axiom is equivalent to an assertion that the class classID is equivalent to the intersection of all of the descriptions. When the modality is partial, it is equivalent to an assertion that the class classID is a subclass of the intersection of all of the descriptions.

Disjointness, equivalence or subsumption relationships can also be asserted with respect to arbitrary class descriptions, and an axiom for naming enumerated classes is also provided:

axiom ::= DisjointClasses "(" **description** {**description**} ")"
| EquivalentClasses "(" **description** {**description**} ")"
| SubClassOf "(" sub=**description** super=**description** ")"
| EnumeratedClass "(" **classID** {**individualID**} ")"

A compound form of property axiom is also provided as follows:

axiom ::= DatatypeProperty "(" **datavaluedPropertyID**
{super=**datavaluedPropertyID**}
{domain=**description**}
{range=**dataRange**}
[Functional] ")"
| ObjectProperty "(" **individualvaluedPropertyID**
{super=**individualvaluedPropertyID**}
{domain=**description**}
{range=**description**}
[inverseOf=**individualvaluedPropertyID**]
[Symmetric]
[Functional | InverseFunctional | Transitive] ")"

The two different forms of axiom again reflect the treatment of datatypes in DAML+OIL. For datavalued properties, all super-properties must also be datavalued properties, an inverse property cannot be specified, the range must be a data range, and they cannot be Symmetric, InverseFunctional or Transitive. For individualvalued properties, all super-properties must also be individualvalued properties, an inverse property can be specified, the range must be a description, and they can be any of Symmetric, Functional, InverseFunctional or Transitive.

As for classes, equivalence and subsumption relationships can also be asserted with respect to properties:

axiom ::= EquivalentProperties "(" **datavaluedPropertyID**
{**datavaluedPropertyID**} ")"
| SubPropertyOf "(" **datavaluedPropertyID**
datavaluedPropertyID ")"
| EquivalentProperties "(" **individualvaluedPropertyID**
{**individualvaluedPropertyID**} ")"
| SubPropertyOf "(" **individualvaluedPropertyID**
individualvaluedPropertyID ")"

Finally, a range of axioms are provided for asserting facts about individuals and their relationships:

fact ::= **individual**
| SameIndividual "(" **individualID** {**individualID**} ")"
| DifferentIndividuals "(" **individualID** {**individualID**} ")"
individual ::= Individual "(" [**individualID**] {type=**description**}
{**propertyValue**} ")"

propertyValue ::= "(" **individualvaluedPropertyID individualID** ")"
| "(" **individualvaluedPropertyID individual** ")"
| "(" **datavaluedPropertyID dataLiteral** ")"

The individual axiom allows an individual to be asserted to be of a given type (i.e., an instance of a class or class description), or to be related to other individuals (via individual valued properties) or data literals (via data valued properties). The SameIndividual and DifferentIndividuals axioms allow one or more individual names to be asserted to refer to the same object or to distinct objects respectively.

5.2 OWL Lite

As well as the above mentioned changes in the language, OWL also introduces a named sub-language called OWL Lite. The objective of owl lite is:

> "[to] provide a language that is viewed by tool builders to be easy enough and useful enough to support". One expectation is that tools will facilitate the widespread adoption of OWL and thus OWL language designers should attempt to create a language that tool developers will flock to. While it is widely appreciated that all of the features in languages such DAML+OIL are important to some users, it is also understood that languages as expressive as DAML+OIL may be daunting to some groups who are trying to support a tool suite for the entire language. In order to provide a target that is approachable to a wider audience, a smaller language has been defined, now referred to as OWL Lite. OWL Lite attempts to capture many of the commonly used features of OWL and DAML+OIL. It also attempts to describe a useful language that provides more than RDFS with the goal of adding functionality that is important in order to support web applications."

OWL Lite differs from OWL in the following respects:

1. It does not provide the oneOf (EnumeratedClass) constructor, so there are no extensionally defined (enumerated) classes.

2. It does not provide the disjointWith (DisjointClasses) constructor, but it is easy to make classes disjoint by using conflicting cardinality restrictions. E.g., if *C* and *D* are classes we can simply introduce a new property *P* and assert that *C* has a minCardinality of 1 with respect to *P* and *D* has a maxCardinality of 0 with respect to *P*. Using the abstract syntax, this can be written as:

   SubClassOf(sub=*C* super=restriction(*P* minCardinality (1)))
   SubClassOf(sub=*D* super=restriction(*P* maxCardinality (0)))

3. It does not support either sameClassAs or subClassOf applied to class expressions. I.e., in statements of the form *C* `subClassOf` *D* (SubClassOf(sub=*C* super=*D*)) and *C* `sameClassAs` *D* (EquivalentClasses(*C D*)), *C* must be a class name and *cannot* be a class expression built using one or more of the constructors described

in Section 4.1. Again, this does not really restrict the expressive power of the language as we can introduce a new class C', assert C' to be equivalent to the desired expression, and then use C' to in the subClass or sameClassAs axiom.

4. It does not support the boolean constructors unionOf, intersectionOf and complementOf. The intersectionOf constructor is, however, implicitly provided by compound descriptions in the abstract syntax.

5. It only supports the use of cardinality restrictions with cardinalities of either 0 (zero) or 1 (one).

5.3 Datatypes

Although the abstract syntax covers the treatment of datatypes in some detail, this treatment follows that of DAML+OIL and may not reflect the treatment of datatypes in the OWL language, which has yet to be determined by the Web Ontology working group.

5.4 Semantics

It is assumed that the semantics of OWL will be similar to those of DAML+OIL, but this is still being debated by the Web Ontology Working Group, some of whose members would like to see an even tighter linkage to RDF, and a relaxation on the some of the restrictions DAML+OIL places on the use of classes, properties, individuals and datatypes.

6 A Tutorial Introduction to OWL

In this section, we provide an introduction to the features of the ontology language. First, however, it is useful to state what this section is **not**. It is not intended as:

- a guide to the *use* of OWL or an exposition of modelling methodologies.
- a detailed description of the semantics of the language. See 4.4 for this;
- a guide to the syntax of OWL ontologies. It is envisaged that the primary means of constructing OWL ontologies will be through tools such as editors (e.g. OilEd or OntoEdit) rather than through manipulation of the raw syntax.

Rather, this section provides an overview description of the various features of the language and illustrates their semantics through the use of a number of examples. It also highlights a number of areas where the interpretation of the language operations are known to cause confusion (for example allValuesFrom as discussed in 6.3.5)

Later versions of this deliverable will incorporate further information and recommendations from other Workpackages such as Workpackages 3, 4 and 5 as and when they become available.

In any such a description, a certain amount of forward reference is required (for example, it is difficult to describe axioms without a certain amount of reference to concept descriptions and vice versa). Hopefully the examples will be kept sufficiently simple to allow the reader to cope with such forward references.

6.1 A Note about Syntax

The RDFS based syntax of languages like OWL can often be a source of confusion. The encoding can produce somewhat "longwinded" concrete syntax which is difficult to read.[11] In order to alleviate this, the examples in this tutorial are given using both OWL "abstract" syntax (see Section 5.1) and the corresponding OWL. In addition, we give simple names (rather than URIs) for all ontology classes, properties and individuals (and which are assumed to occur in the same namespace to ease readability).

6.2 Ontologies in OWL

OWL provides a language for defining ontologies, where an ontology describes/constrains the various terms (classes, properties and individuals) in a specific domain of discourse.

6.2.1 Classes, Taxonomies and Properties

The interpretation of an OWL ontology[12] involves a collection of objects or instances, known as the *domain*. These instances can be organised into collections known as *classes*. In particular, these classes can be described in terms of the *properties* of the individuals that make up the class.

Most uses of an ontology depend ultimately upon the ability to reason about individuals from the domain. In order to do this in a useful fashion we need to have a mechanism to describe the classes that individuals belong to and the properties that they inherit by virtue of class membership. We can always assert specific properties about individuals, but much of the power of ontologies comes from class-based reasoning.

A crucial relationship within an ontology is that of *subclass*. A class B is said to be a subclass of another class A when it is necessarily the case that all instances of B *must* be instances of A. This *taxonomic* relationship can come about either through the direct assertion of the subclass relationship, or through some inference process based on the intensional properties of the classes. The properties let us assert general facts about the members of classes and specific facts about individuals.

In the context of OWL, a property is a binary relation. Two types of properties are distinguished:

- object properties, relations between elements of two classes;
- datatype properties, relations between elements of classes and datatypes;

Descriptions or definitions of particular classes can be given in terms of other classes and properties in the ontology (using the OWL operators). The semantics of OWL then provide a formal description as to when individuals are instances of classes – we can make use of a reasoner in order to infer additional information or relationships between classes, in particular inferring taxonomic relations, equivalences or inconsistencies.

[11] This is not a major issue though if we consider that the underlying concrete syntax, e.g. the XML/RDFS encoding is not intended to be a format directly manipulated by humans.

[12] This should not be taken as a detailed description of the semantics of OWL – such a detailed exposition is given in 4.4.

6.2.2 Necessary and Sufficient Conditions

It is important when modelling with languages such as OWL to be aware of the difference between *necessary* and *sufficient* conditions for class membership. In traditional frame based languages and systems, classes are usually described in terms of *necessary* conditions. These describe properties of the individuals that must hold if the individual is a member of the particular class. In OWL, these necessary conditions are represented using the subClassOf axiom. For example, we can assert that Dog is a subclass of Mammal:

```
SubClassOf (Dog Mammal)
```

```
<owl:Class rdf:ID="Dog">
  <rdfs:subClassOf rdf:resource="#Mammal"/>
</owl:Class>
```

This says that in order to be an instance of the class Dog, an individual must also be an instance of class Mammal. However, there may be instances of the class Mammal that are not Dogs. Thus being a Mammal is a *necessary* condition of being a Dog, but is not *sufficient*.

In contrast, *sufficient* conditions tell us precisely those properties an individual must have in order to be recognised as a member of the class. Sufficient conditions can be expressed through the use of sameClassAs axioms. For example, we can define CarOwner as precisely those Persons that own Cars:

```
EquivalentClasses (
 CarOwner
 intersectionOf(Person restriction(owns someValuesFrom = Car))
)
```

```
<owl:Class rdf:ID="CarOwner">
  <owl:sameClassAs>
    <owl:Class>
      <owl:intersectionOf rdf:parsetype="owl:collection">
        <owl:Class rdf:about="Person"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#owns"/>
          <owl:someValuesFrom rdf:resource="#Car"/>
        </owl:Restricton/>
      </owl:intersectionOf>
    </owl:Class>
  </sameClassAs>
<owl:Class>
```

See below for further details of how to interpret the expressions used in this example.

In this case, the definition says that all CarOwners must be a Person who owns a Car (necessary conditions as above), but in addition, any Person who owns a Car must

also be a CarOwner. This is a much stronger assertion than simply giving the necessary conditions. The provision of *defined* classes through necessary and sufficient conditions is one of the key aspects of this kind of language approach that distinguishes it from frame systems, and allows us to employ a reasoner to infer additional information and relationships.

Of course, we can provide a mixture of necessary and sufficient conditions. We can extend the example by asserting that:

SubClassOf (CarOwner Adult)

```
<owl:Class rdf:ID="CarOwner">
  <owl:subClassOf rdf:resource="#Adult">
</owl:Class>
```

Here we are saying that all car owners must be adults. The fact that car owners are adults is not a sufficient condition for class membership though, so we need not know *a priori* that an individual is a Adult before recognising them as an CarOwner. However, once the individual has been recognised as a CarOwner, the inference can be made.

6.3 Class Constructors

The language provides a range of constructors for building complex class descriptions from simpler ones. Here we provide an informal description of the semantics of these constructors, and examples of how the constructors might be used.

Boolean Connectives (intersectionOf, unionOf and complementOf), allow us to combine class descriptions using the familiar logical connectives. *Restrictions* (someValuesFrom, allValuesFrom, minCardinality, maxCardinality, cardinality) allow us to describe the kinds of relationship that individuals in a certain class must participate in. *Enumerations* (the oneOf constructor) allow us to provide explicit extensional definitions of a class.

6.3.1 intersectionOf

The intersectionOf constructor combines two classes, forming their intersection. For example, we may wish to describe the set of individuals that are both Women and Drivers (i.e., women drivers):

intersectionOf (Woman Driver)

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="owl:collection">
    <owl:Class rdf:about="Woman"/>
    <owl:Class rdf:about="Driver"/>
  </owl:intersectionOf>
</owl:Class>
```

The individuals which are members of this class are precisely those individuals which are members of both the class Woman and the class Driver.

6.3.2 unionOf

The unionOf constructor combines two classes, forming their union. For example, we may wish to describe the set of individuals that are either Cats or Dogs (i.e., common domestic pets):

unionOf (Cat Dog)

```
<owl:Class>
  <owl:unionOf rdf:parseType="owl:collection">
    <owl:Class rdf:about="Cat"/>
    <owl:Class rdf:about="Dog"/>
  </owl:unionOf>
</owl:Class>
```

The individuals which are members of this class are then precisely those individuals which are either members of the class Cat or the class Dog.

Note that we must be very careful with disjunction and conjunction when talking about these classes using natural language. When we use a phrase such as “Cats and Dogs”, the intuitive interpretation of this is actually all those individuals that are either Cats or Dogs, e.g. in formal OWL terms, the class (unionOf Cat Dog). This is often a source of confusion. Similarly, the natural language interpretation of “or” is often as an exclusive or — although logically correct, providing the answer “yes” to the question “would you like coffee or tea?” is generally considered unhelpful.

6.3.3 complementOf

The complementOf constructor is applied to a single class and describes the collection of individuals which are known not to be instances of the class. For example, we may wish to describe the set of individuals who only eat things which are not animal products (i.e., vegetarians).

restriction (eats allValuesFrom = complementOf (AnimalProduct))

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats"/>
  <owl:allValuesFrom>
    <owl:Class>
      <owl:complementOf rdf:resource="#AnimalProduct"/>
    </owl:Class>
  </owl:allValuesFrom>
</owl:Restriction>
```

Open and Closed Worlds

The complementOf constructor highlights an interesting issue relating to the semantics of OWL. The standard semantics of OWL (see Section 4.4) makes an *open world assumption* (OWA). This means that we cannot assume that all information is known about all the individuals in the domain. Thus, simply being unable to prove that an individual a is an instance of X does not justify our concluding that a is not an instance of X.

This is in contrast to languages or systems using *negation as failure* or a *closed world assumption* (CWA). In this case, if we cannot deduce that an individual a is an instance X then we can assume that a is an instance of (complementOf X).

The OWA facilitates reasoning about intensional definitions of classes — we need not know all the information about the world in order to be able to make deductions about the relationships between classes. In contrast, the CWA may facilitate reasoning about a particular state of affairs.

The ramifications of the OWA can sometimes be confusing. Note, for example, that even in the presence of the OWA, we can always deduce that for any individual a, a must be an instance of:

unionOf (A complementOf (A))

i.e. a must be in either A or its complement. It may be the case, however, that we cannot determine (given the information at our disposal) exactly which of the two it is an instance of. Although an OWL reasoner should always answer yes to the question "is a an instance of (unionOf (A (complementOf A)))", it may answer no to both "is a an instance of A" *and* "is a an instance of (complementOf A)".

For a further illustration of this see the discussion of domain and range restrictions in 6.6.3.

6.3.4 someValuesFrom

The someValuesFrom constructor describes those individuals that have a relationship to other individuals of a particular class. For example, we may wish to describe the collection of individuals that are car owners, i.e. all those things that own a Car.

restriction (owns someValuesFrom = Car)

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#owns"/>
  <owl:someValuesFrom rdf:resource="#Car"/>
</owl:Restriction>
```

The individuals that are members of this class are those that are related via the owns property to at least one instance of the Car class.

Note that the someValuesFrom constructor makes no restriction about other relationships that may be present. So an individual of the class described above may own many Cars, and may also own other objects which are not Cars.

6.3.5 allValuesFrom

The allValuesFrom constructor describes those individuals with relationships that are restricted to a particular class. For example, the collection of individuals that only own cars is described as:

restriction (owns allValuesFrom = Car)

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#owns"/>
  <owl:allValuesFrom rdf:resource="#Car"/>
</owl:Restriction>
```

The individuals that are members of this class are those such that if there is an object that is related to them via the owns property, then it *must* be a Car.

Note that the allValuesFrom constructor makes no assertion about the existence of the relationship, simply that if the relationship holds, then the related object must be of the particular class. This is often a source of confusion with the allValuesFrom constructor. As an example of this, consider the class description:

intersectionOf (Person restriction (children allValuesFrom = Doctor))

```
<owl:Class>
  <owl:intersectionOf rdf:parsetype="owl:collection">
    <owl:Class rdf:about="Person"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#children"/>
      <owl:allValuesFrom rdf:resource="#Doctor"/>
    </owl:Restricton/>
  </owl:intersectionOf>
</owl:Class>
```

This describes the collection of Persons, all of whose children are Doctors. If we have an individual Sean who is *known to have no children*, then we can infer that Sean will be a member of this class.

6.3.6 minCardinality

The minCardinality constructor describes those individuals that participate in a particular relationship with at least a minimum number of distinct individuals. For example:

restriction (children minCardinality(2))

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#children"/>
  <owl:minCardinality>2</owl:minCardinality>
</owl:Restricton/>
```

describes all those individuals that have at least 2 children. Note that a minCardinality restriction makes no assertion about the maximum number of individuals that members of the class may be related to via the given property.

6.3.7 maxCardinality

The maxCardinality constructor describes those individuals that participate in a particular relationship with at most a maximum number of distinct individuals. For example:

restriction (children maxCardinality(2))

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#children"/>
  <owl:maxCardinality>2</owl:maxCardinality>
</owl:Restricton/>
```

describes all those individuals that have at most 2 children. Note that a maxCardinality restriction makes no assertion about the minimum number of children that members of the class may be related to via the given property. In particular, individuals that are known not to be related via the given property will be members of a maxCardinality class. Following the example given in 6.3.5, Sean will be a member of this class as he is known to have *no* children.

Note that OWL does *not* make a unique name assumption (UNA). We cannot assume that different names refer to distinct individuals, so Einstein and AlbertEinstein, for example, may be two different names for the same individual. This is often a source of confusion with respect to the maxCardinality constructor as it can lead to inferences about names referring to the same individual rather than to inferences that the ontology is incoherent (contains inconsistent facts). For example, if the individual Sean is asserted to be a member of the above class, *and* is asserted to have children John, Einstein and AlbertEinstein, then we can infer that (at least) two of John, Einstein and AlbertEinstein are different names for the same individual. The ontology is only incoherent if we can prove that this cannot be the case, e.g., if we have asserted that they are different individuals (see Section 6.5).

6.3.8 cardinality

This is provided as a convenience for stating that a property has both a minimum and maximum cardinality, e.g.

restriction (children cardinality(2))

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#children"/>
  <owl:cardinality>2</owl:cardinality>
</owl:Restricton/>
```

describes the class of things with *exactly* 2 children.

Local vs. Global

Note that someValuesFrom, allValuesFrom, and the cardinality restrictions are *local* restrictions that apply only to the class being described. The example above concerning Persons all of whose children are Doctors is not making an assertion about the permissible values for the children property in general. In order to do that, we can use domain and range restrictions on properties, which then apply globally (See 6.6.2 and 6.6.3).

6.3.9 oneOf

The constructors described above allow us to provide *intensional* definitions of classes, that is definitions of the classes through the properties that instances of the classes have. In contrast, the oneOf constructor allows us to specify classes *extensionally*, i.e. in terms of an enumeration of the individuals that are instances of the class. The oneOf operation is quite strong as it completely defines the class. No other individuals (that were not explicitly enumerated) can be instances of the class (unless, of course, they are the same as one of the individuals enumerated – see 6.5.1). Enumerations allow us to describe, for example, the collection of DaysOfTheWeek:

EnumeratedClass (DaysOfTheWeek Mon Tue Wed Thu Fri Sat Sun)

```
<owl:Class rdf:ID="DaysOfTheWeek">
  <owl:sameClassAs>
    <owl:oneOf rdf:parseType="owl:collection">
      <owl:Thing rdf:about="#Mon"/>
      <owl:Thing rdf:about="#Tue"/>
      <owl:Thing rdf:about="#Wed"/>
      <owl:Thing rdf:about="#Thu"/>
      <owl:Thing rdf:about="#Fri"/>
      <owl:Thing rdf:about="#Sat"/>
      <owl:Thing rdf:about="#Sun"/>
    </owl:oneOf>
  </owl:sameClassAs>
</owl:Class>
```

6.3.10 hasValue

The hasValue constructor describes those individuals which have a particular given value for a property. For example, Italians can be defined as:

```
EquivalentClasses (
  Italian
  intersectionOf (Person restriction (bornIn value = Italy))
)
```

```
<owl:Class rdf:ID="Italian">
  <owl:sameClassAs>
```

```
    <owl:Class>
      <owl:intersectionOf rdf:parsetype="owl:collection">
        <owl:Class rdf:about="Person"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#bornIn"/>
          <owl:hasValue rdf:resource="#Italy"/>
        </owl:Restricton/>
      </owl:intersectionOf>
    </owl:Class>
  </sameClassAs>
<owl:Class>
```

where Italy is an individual from the domain. The hasValue is equivalent to a someValuesFrom restriction where the class description is an enumeration (see 6.3.9) containing a single individual.

6.4 Class Axioms

Class axioms in OWL assert relationships between classes. For example, we can provide subClassOf axioms, which assert subclass relationships (and thus *necessary* conditions as described in 6.2.2). We can also give sameClassAs axioms asserting equivalence between classes (and thus *necessary* and *sufficient* conditions). In addition, we can assert that two (or more) classes are *disjoint*, i.e. that an individual cannot be an instance of both (or any two of the) classes.

6.4.1 subClassOf

The subClassOf axiom asserts a relationship between two classes, specifically that all the instances of one must be instances of another. The two classes could simply be primitive classes, or could involve more complex expressions. For example, we can assert that:

SubClassOf (Dog Mammal)

```
<owl:Class rdf:ID="Dog">
  <rdfs:subClassOf rdf:resource="#Mammal"/>
</owl:Class>
```

This states that all instances of Dog are also instances of Mammal. A slightly more complex example is:

```
SubClassOf (
 Sheep
 intersectionOf (Animal (eats allValuesFrom = Grass))
)
```

```
<owl:Class rdf:ID="Sheep">
  <rdfs:subClassOf>
    <owl:intersectionOf rdf:parsetype="owl:collection">
      <owl:Class rdf:about="Animal"/>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#eats"/>
        <owl:allValuesFrom rdf:resource="#Grass"/>
      </owl:Restricton/>
    </owl:intersectionOf>
  </rdfs:subClassOf>
</owl:Class>
```

asserting that Sheep are animals that only eat Grass. A third example is:

```
SubClassOf (
 restriction (drives someValuesFrom = Bus)
 restriction (reads allValuesFrom = Tabloid)
)
```

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#drives"/>
  <owl:someValuesFrom rdf:resource="#Bus"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#reads"/>
      <owl:allValuesFrom rdf:resource="#Tabloid"/>
    </owl:Restricton/>
  </rdfs:subClassOf>
</owl:Restricton/>
```

the somewhat sweeping generalisation that anything that drives a Bus only reads Tabloids[13]. Note that here the "left hand side" of the subclass assertion is not simply a class name, but is in fact a composite description.

6.4.2 sameClassAs

A sameClassAs axiom asserts equivalence between two classes, i.e. any instance of one must be an instance of the other and vice versa. Again, as with the subClassOf, the classes could be primitives or complex descriptions. A sameClassAs:

```
EquivalentClasses (Man Bloke)
```

```
<owl:Class rdf:ID="Man">
  <owl:sameClassAs rdf:resource="#Bloke"/>
</owl:Class>
```

[13] A *tabloid* being a newspaper that is about half the page size of an ordinary newspaper and that contains news in condensed form and much photographic matter. [Merriam-Webster]

is equivalent to a pair of subClassOf axioms

SubClassOf (Man Bloke)
SubClassOf (Bloke Man)

6.4.3 disjointWith

A disjointWith axiom asserts that the extensions of the classes involved are disjoint. Thus the assertion:

DisjointClasses (Cat Dog)

```
<owl:Class rdf:ID="Cat">
  <owl:disjointWith rdf:resource="#Dog"/>
</owl:Class>
```

tells us that the collection of instances of Cat and the collection of instances of Dog are disjoint – i.e. there can be no individual x s.t. x is both a Cat and a Dog.

6.5 Individual Axioms

Individual axioms allow us to assert facts about particular individuals.

6.5.1 sameIndividualAs

Two individuals can be asserted to be the same. This allows us to refer to the same individual using two different names.

SameIndividual (SeanKBechhofer SeanBechhofer)

```
<owl:Individual rdf:ID="SeanKBechhofer">
  <owl:sameIndividualAs rdf:about="#SeanBechhofer"/>
</owlIndividual>
```

6.5.2 differentIndividualFrom

Two individuals can be asserted to be distinct.

DifferentIndividuals (Tibbs Ginger)

```
<owl:Individual rdf:ID="Tibbs">
  <owl:differentIndividualFrom rdf:about="#Ginger"/>
</owlIndividual>
```

In the light of the above assertion, if we know that John owns Tibbs and owns Ginger (and that Tibbs and Ginger are Cats), then we can infer that John owns at least 2 Cats.

6.6 Property Axioms

Property axioms assert general characteristics of properties such as property hierarchies, global ranges and domains. On introduction, properties should be identified as being object properties (ObjectProperty) or data type properties (DatatypeProperty).

6.6.1 subPropertyOf

Property hierarchies can be created by stating that some properties are subproperties of other properties. For example:

ObjectProperty (hasPet super = owns)

```
<owl:ObjectProperty rdf:ID="hasPet">
  <rdfs:subPropertyOf rdf:resource="#owns"/>
</owl:ObjectProperty>
```

This states that hasPet is a subproperty of owns. From this a reasoner may deduce that if John is related to Tibbs by the hasPet property, then John is also related to Tibbs by the owns property.

6.6.2 domain

A domain axiom for a property r states that for any two individuals x and y such that x is related to y via r, then the individual x must be an instance of the given class. For example:

ObjectProperty (owns domain = Person)

```
<owl:ObjectProperty rdf:ID="owns">
  <rdfs:domain rdf:resource="#Person"/>
</owl:ObjectProperty>
```

states that any individual that owns something must be a Person. Note that in contrast to the someValuesFrom local restriction, the domain assertion makes no claim about the existence of the relationship. In our example, it is simply the case that if there *is* an individual that owns something then it must be a Person. Similarly, this is not an assertion that all Persons must own something — the assertion is quite consistent with a world where nothing is owned or where there is a Person who owns nothing.

6.6.3 range

A range axiom for a property r states that any for any two individuals x and y such that x is related to y via r, then the individual y must be an instance of the given class. For example:

ObjectProperty (eats range = Food)

```
<owl:ObjectProperty rdf:ID="eats">
  <rdfs:range rdf:resource="#Food"/>
</owl:ObjectProperty>
```

This says that the only things that can be eaten are instances of Food. Again, as with the domain restriction, the range makes no claim as to the existence of the relationship. The assertion above is quite consistent with a world where nothing eats anything, or where there are instances of Food which are not eaten.

Inference and domain and range

The domain and range assertions differ from someValuesFrom and allValuesFrom in that they are global assertions that apply *everywhere* the property is used. As such that are strong assertions.

The use of domain and range restrictions can occasionally cause unexpected side effects in terms of inference. For example, if we have asserted that:

ObjectProperty (owns domain = Person)
Individual(Mike (owns Tibbs))

but have not explicitly stated that Mike is a Person, a reasoner can deduce that this is indeed the case (i.e. that Mike is an instance of Person). This is sometimes considered unexpected behaviour by some users, who would rather expect this to be an error as we have not explicitly stated that Mike is a Person.

6.6.4 inverseOf

If r is the inverse of s, then whenever x is related to y via r, then y is related to x via s. The inverseOf assertion allows us to state this:

ObjectProperty (owns inverseOf = ownedBy)

```
<owl:ObjectProperty rdf:ID="owns">
  <owl:inverseOf rdf:resource="#ownedBy"/>
</owl:ObjectProperty>
```

Thus if Fred owns Tibbs, then we know that Tibbs is ownedBy Fred.

6.6.5 FunctionalProperty

The FunctionalProperty assertion states that a property is functional — if a particular individual has a value for the property, that value must be unique. For example, we might state:

ObjectProperty (ownedBy Functional)

```
<owl:ObjectProperty rdf:ID="ownedBy">
  <rdf:type>
    <owl:FunctionalProperty/>
  </rdf:type>
</owl:ObjectProperty>
```

This says that any individual can have at most one owner — if we find assertions that say:

Individual (Tibbs (ownedBy Fred))
Individual (Tibbs (ownedBy Jim))

then a reasoner can deduce that Fred and Jim must be the same individual.

The FunctionalProperty assertion can be seen as a kind of global cardinality constraint that says for any use of the property, the minimum cardinality is 0 and the maximum cardinality is 1.

6.6.6 TransitiveProperty

If a property r is transitive, then whenever x is related to y via r and y is related to z via r, then it must be the case that x is also related to z via r. The TransitiveProperty assertion asserts that a property is transitive. An example of a transitive property is ancestor. Any ancestors of my ancestors are also ancestors of mine.

ObjectProperty (ancestor Transitive)

```
<owl:ObjectProperty rdf:ID="ancestor">
  <rdf:type>
    <owl:FunctionalProperty/>
  </rdf:type>
</owl:ObjectProperty>
```

Due to the desire to retain *decidability*[14] (and thus allow the use of a reasoner), there are a number of conditions that restrict the interaction of functional and transitive properties. These effectively boil down to the constraint that a transitive property (or any superproperty of a transitive property) should not be used in a cardinality constraint. If this restriction is violated, then checking class consistency may no longer be decidable (see [13] for details as to why this is the case).

Note that as FunctionalProperty can be considered as a cardinality constraint, this means that transitive roles (and their superproperties) cannot be declared functional (if decidability is to be retained).

[14] A reasoning problem is said to be *decidable* if there is a computational process (e.g. a computer program) that solves the problem in a finite number of steps, i.e. the program *terminates*.

6.6.7 SymmetricProperty

A symmetric property is one where whenever x is related to y via r then it must also be the case that y is related to x via r. For example, we could say that:

ObjectProperty (friend Symmetric)

```
<owl:ObjectProperty rdf:ID="friend">
  <rdf:type>
    <owl:SymmetricProperty/>
  </rdf:type>
</owl:ObjectProperty>
```

expressing the fact that the friend property is symmetric. Stating that a property is symmetric is equivalent to stating that a property is equal to its own inverse.

6.6.8 InverseFunctionalProperty

Asserting that a property is inverse functional says that the inverse of the property is functional. This means that knowing the value of the property for a particular individual is sufficient to uniquely identify the individual. For example, we could assert:

ObjectProperty (isMotherOf InverseFunctional)

```
<owl:DataProperty rdf:ID="isMotherOf">
  <rdf:type>
    <owl:InverseFunctionalProperty/>
  </rdf:type>
</owl:DataProperty>
```

This tells us that if Sean is an individual, then no two distinct individuals can be the mother of Sean — if two individuals x and y are found to be the mother of Sean, then we can infer that x and y must, in fact, be different names for the same individual.

Asserting that a property is inverse functional is, of course, equivalent to asserting that its inverse property is functional. E.g., if hasMother is the inverse of isMotherOf, then:

ObjectProperty (hasMother Functional)

is equivalent to the above inverse functional assertion with respect to isMotherOf.

6.7 Datatypes

The final OWL recommendation will be based on XML Schema datatypes. As discussed above, the datatype and “object” domains will (probably) be considered distinct. This version of the tutorial does not discuss datatypes, but will be extended to cover that area of the language once a firm proposal is clear.

7 A Worked Example

In this section we provide an example ontology that demonstrates the various OWL constructors and axioms in use. We also illustrate the inferences that a reasoner can make based on the descriptions of the classes. This section may be revised in subsequent versions of the deliverable to incorporate material, guidelines and recommendations from other workpackages including WPs 3, 4 and 5. Again, note that this is not intended as a guide to modelling and building ontologies in OWL, but is rather an illustration of the way that class definitions may interact.

7.1 Basic Classes

To begin with, we introduce a number of basic classes such as Animal, Person, Vehicle.

```
<owl:Class rdf:ID="#Animal"/>
<owl:Class rdf:ID="#Person"/>
<owl:Class rdf:ID="#Vehicle"/>
```

All this does is introduce names for some classes — it says nothing about the instances (if any) that the classes might have or any properties of those instances. We can further refine this by introducing a subclass axiom:

SubClassOf (Person Animal)

```
<owl:Class rdf:about="#Person">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Animal"/>
 </rdfs:subClassOf>
</owl:Class>
```

This then tells us that all Persons are also Animals. We can, of course, combine the introduction of classes with some additional information, e.g., about their (subclass) relationship to other classes:

SubClassOf (Cat Animal)
SubClassOf (Dog Animal)
SubClassOf (Bus Vehicle)
SubClassOf (Car Vehicle)
SubClassOf (Lorry Vehicle)

```
<owl:Class rdf:ID="#Cat">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Animal"/>
 </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="#Dog">
```

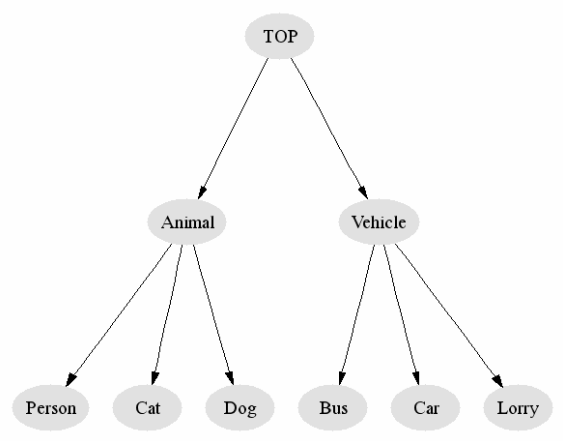

Figure 5: Simple Taxonomy

```
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Animal"/>
 </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="#Bus">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Vehicle"/>
 </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="#Car">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Vehicle"/>
 </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="#Lorry">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Vehicle"/>
 </rdfs:subClassOf>
</owl:Class>
```

At this point, we have a very simple taxonomic structure as shown in Figure 5.

7.2 Properties

We now introduce a property that we will use to relate instances in the world.

ObjectProperty (owns domain = Person)

```
<owl:ObjectProperty rdf:ID="#owns">
```

```
 <rdfs:domain>
  <owl:Class rdf:about="#Person"/>
 </rdfs:domain>
</owl:ObjectProperty>
```

The owns property is asserted as having a domain of Person – this means that only Persons can own things. A subproperty of owns is hasPet:

```
ObjectProperty (hasPet
 super = owns
 range = Animal)
```

```
<owl:ObjectProperty rdf:ID="#hasPet">
 <rdfs:subPropertyOf rdf:resource="#owns"/>
 <rdfs:range>
  <owl:Class rdf:about="#Animal"/>
 </rdfs:range>
</owl:ObjectProperty>
```

As hasPet is a subproperty of owns (see 6.6.1), it will inherit the domain restriction[15]. We have also placed a range restriction (see 6.6.3) on hasPet here, saying that the only things that can be pets are Animals.

We can also define inverse (see 6.6.4) properties:

```
ObjectProperty (ownedBy
 inverseOf = owns
 Functional)

ObjectProperty (isPetOf
 inverseOf = hasPet)
```

```
<owl:ObjectProperty rdf:ID="#ownedBy">
 <owl:inverseOf rdf:resource="#owns"/>
 <rdf:type>
   <owl:FunctionalProperty/>
 </rdf:type>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="#isPetOf">
 <owl:inverseOf rdf:resource="#hasPet"/>
</owl:ObjectProperty>
```

Here we have also stated that the ownedBy property is functional (see 6.6.5) – thus any one thing can only ever have a single owner (but of course need not have one).

[15]Why is this? If John hasPet Tibbs, then we know that John owns Tibbs, and thus that John must be a Person due to the domain restriction on owns.

7.3 Defining Classes

Now we can start to define classes via their properties.[16] For example:

```
EquivalentClasses (
 CatOwner
 restriction (hasPet someValuesFrom = Cat)
)
```

```
<owl:Class rdf:ID="#CatOwner">
 <owl:sameClassAs>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasPet"/>
   <owl:someValuesFrom>
    <owl:Class rdf:about="#Cat"/>
   </owl:someValuesFrom>
  </owl:Restriction>
 </owl:sameClassAs>
</owl:Class>
```

This tells us that the class of Cat owners are *exactly* those instances that are related to an instance of the class Cat via the hasPet property. Similarly we can define a Dog owners as those who have Dogs as pets.

```
EquivalentClasses (
 DogOwner
 restriction (hasPet someValuesFrom = Dog)
)
```

```
<owl:Class rdf:ID="#DogOwner">
 <owl:sameClassAs>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasPet"/>
   <owl:someValuesFrom>
    <owl:Class rdf:about="#Dog"/>
   </owl:someValuesFrom>
  </owl:Restriction>
 </owl:sameClassAs>
</owl:Class>
```

At this point, a reasoner will be able to make some inferences concerning the class definitions that we have produced. For example, due to the domain restriction on hasPet (see above), we can infer that any Dog Owner must be a Person. If we now introduce a more generic notion of a Pet Owner:

[16] Note that there is no requirement for classes to have a single definition (or any definition at all). A triangle could, for example, be asserted to be equivalent to the class of polygons with exactly 3 sides *and* to be equivalent to the class of polygons with exactly 3 angles (see Section 6.2.2).

```
EquivalentClasses (
 PetOwner
 restriction (hasPet someValuesFrom = Thing)
)
```

```
<owl:Class rdf:ID="#PetOwner">
 <owl:sameClassAs>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasPet"/>
   <owl:someValuesFrom>
    <owl:Thing/>
   </owl:someValuesFrom>
  </owl:Restriction>
 </owl:sameClassAs>
</owl:Class>
```

we can similarly now deduce that all PetOwners are Persons. In addition, we can also infer that all DogOwners and all CatOwners are PetOwners.

To demonstrate cardinalities, we can define an AnimalLover as:

```
EquivalentClasses (
 AnimalLover
 restriction (hasPet minCardinality(3))
)
```

```
<owl:Class rdf:ID="#AnimalLover">
 <owl:sameClassAs>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasPet"/>
   <owl:minCardinality>3</owl:minCardinality>
  </owl:Restriction>
 </owl:sameClassAs>
</owl:Class>
```

Thus an AnimalLover is anything that has at least 3 pets[17]. Again the domain restriction allows us to deduce that an AnimalLover is a PetOwner – anything that has at least 3 pets, must have one.

[17]This definition illustrates a crucial point. An ontology represents a particular view of the world. It may be that different opinions exists as to what constitutes an AnimalLover. For example, you may claim that it is not necessary to have any pets in order to be considered an AnimalLover. This, however is an issue concerned primarily with how one models the domain and uses the language, rather than with the language itself. You may well disagree with *my* definition of what an AnimalLover is, but the explicit definition given here makes it clear what *I* consider an AnimalLover to be (for the purposes of this ontology). Languages such as OWL give us a rich palette of operations that facilitate the capture of our intended meanings.

7.4 More Complex Inference

Next, we introduce two new subclasses of Person:

SubClassOf (Woman Person)
SubClassOf (OldLady Woman)

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

<owl:Class rdf:ID="#OldLady">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Woman"/>
 </rdfs:subClassOf>
</owl:Class>
```

So far, we are not stating any of the properties of these classes, other than simple taxonomic relationships. We also introduce two axioms concerning these classes:

SubClassOf (Woman restriction (hasPet someValuesFrom = Thing))

```
<owl:Class rdf:about="#Woman">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasPet"/>
   <owl:someValuesFrom>
    <owl:Thing/>
   </owl:someValuesFrom>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Note that, although it is similar to the axiom relating to Pet Owner, this particular axiom uses the subClassOf operation, rather than sameClassAs. Thus we can infer that in our world, all instances of Woman must have pets,[18] but not that anyone who has a pet is a Woman.

SubClassOf (OldLady restriction (hasPet allValuesFrom = Cat))

[18]This example model is not necessarily intended to represent the *true state* of the world, but is simply an illustrative example.

```
<owl:Class rdf:about="#OldLady">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasPet"/>
   <owl:allValuesFrom>
    <owl:Class rdf:about="#Cat"/>
   </owl:allValuesFrom>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

This particular axiom then constrains the class OldLady, saying that the *only* things that an OldLady can have as pets are Cats (see 6.3.5). This in itself does not say anything about the fact that an OldLady has a pet. Taken with the rest of the stated axioms, however, we can infer that an OldLady is a Woman, thus must have a pet, and this then allows us to infer that an OldLady must be a Cat Owner.

Next we can state that Dogs and Cats are disjoint (i.e. there is no instance in the world that is both a Cat and a Dog).

```
DisjointClasses (Cat Dog)
```

```
<owl:Class rdf:about="#Cat">
 <owl:disjointWith>
  <owl:Class rdf:about="#Dog"/>
 </owl:disjointWith>
</owl:Class>
```

Now by using a combination of an allValuesFrom and a complementOf, we can introduce the concept of a Person who definitely does not have a Dog:

```
EquivalentClasses (
 NotDogOwner
 intersectionOf (
  Person
  restriction (hasPet allValuesFrom = complementOf (Dog))
 )
)
```

```
<owl:Class rdf:about="#NotDogOwner">
 <owl:sameClassAs>
  <owl:Class>
   <owl:intersectionOf rdf:parseType="owl:collection">
    <owl:Class rdf:about="#Person"/>
     <owl:Restriction>
      <owl:onProperty rdf:resource="#hasPet"/>
      <owl:allValuesFrom>
```

```
        <owl:Class>
         <owl:complementOf>
          <owl:Class rdf:about="#Dog"/>
         </owl:complementOf>
        </owl:Class>
       </owl:allValuesFrom>
      </owl:Restriction>
     </owl:Class>
    </owl:intersectionOf>
   </owl:Class>
  </owl:sameClassAs>
</owl:Class>
```

This rather more complicated assertion tells us that an instance of NotDogOwner must be an instance of Person, and in addition, anything that the instance is related to via hasPet must *not* be an instance of Dog – in other words, this class represents those Persons who are definitely not Dog Owners.

Again, further inferences can now be drawn given these facts. We can infer that OldLady is a subclass of Not Dog Owner as an OldLady can *never* own a Dog. This is due to the allValuesFrom restriction that applies to hasPet on OldLady, along with the disjointness axiom concerning Cats and Dogs. An extended taxonomy taking this into account is shown in Figure 6.

7.5 Individuals

We now introduce a number of individuals, and using basic RDF assertions, make some assertions about their types (i.e., the classes that they are instances of) and the relationships that hold between them.

Individual (Bill type = Person)
Individual (Ted type = Person)

```
<owl:Individual rdf:ID="Bill">
  <rdf:type rdf:resource="#Person"/>
</owl:Individual>
<owl:Individual rdf:ID="Ted">
  <rdf:type rdf:resource="#Person"/>
</owl:Individual>
```

Individual (Tibbs type = Cat)
Individual (Ginger type = Cat)
Individual (Fido type = Dog)

```
<owl:Individual rdf:ID="Tibbs">
  <rdf:type rdf:resource="#Cat"/>
</owl:Individual>
```

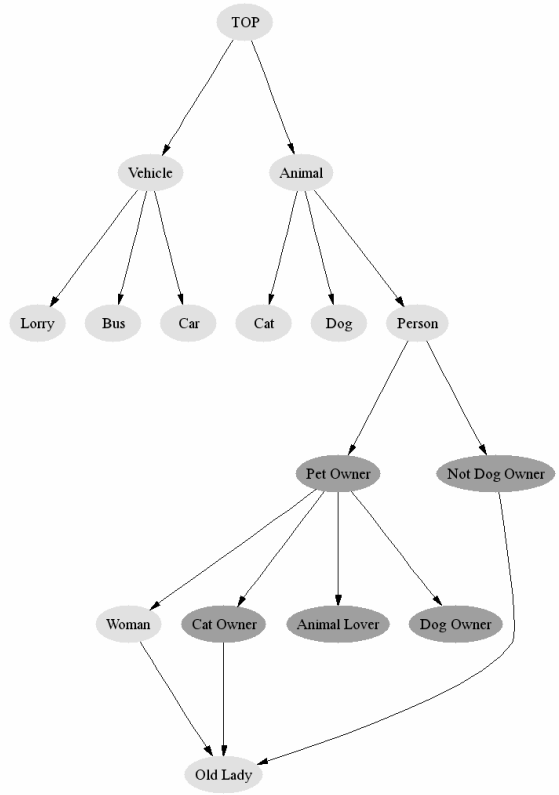

Figure 6: Extended Taxonomy

```
<owl:Individual rdf:ID="Ginger">
  <rdf:type rdf:resource="#Tibbs"/>
</owl:Individual>
<owl:Individual rdf:ID="Fido">
  <rdf:type rdf:resource="#Dog"/>
</owl:Individual>
```

Thus Bill and Ted are Persons, Tibbs and Ginger are Cats, and Fido is a Dog. Further, we can say things about who has which animal as a pet:

```
Individual (Bill (hasPet Tibbs))
Individual (Ted (hasPet Ginger))
```

```
<rdf:Description rdf:about="#Bill">
  <hasPet rdf:resource="#Tibbs"/>
</rdf:Description>
<rdf:Description rdf:about="#Ted">
  <hasPet rdf:resource="#Ginger"/>
</rdf:Description>
```

7.6 Enumerations

An extensionally defined class (see 6.3.9) can be used to explicitly enumerate all the instances of the class. Thus we can define:

```
EquivalentClasses (
 BillOrTedPet
 restriction (isPetOf someValuesFrom oneOf (Bill Ted))
)
```

```
<owl:Class rdf:ID="BillOrTedPet">
 <owl:sameClassAs>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#isPetOf"/>
   <owl:someValuesFrom>
    <owl:Class>
     <owl:oneOf parseType="owl:collection">
      <owl:Thing rdf:about="#Bill"/>
      <owl:Thing rdf:about="#Ted"/>
     </owl:oneOf>
    </owl:Class>
   </owl:someValuesFrom>
  </owl:Restriction>
 </owl:sameClassAs>
</owl:Class>
```

This defines the class BillOrTedPet as being anything that isPetOf one of the individuals in the collection {Bill, Ted}, e.g. a pet of Bill or Ted's. The enumeration allows us to talk about the class that consists of Bill and Ted and use it in class definitions without having to explicitly provide an intensional definition of the class.

Bill and Ted are friends:

```
ObjectProperty (friend domain = Person range = Person Symmetric)
Individual (Bill (friend Ted))
```

```
<owl:ObjectProperty rdf:about="#friend">
 <rdfs:domain>
  <owl:Class rdf:about="#Person"/>
 </rdfs:domain>
 <rdfs:range>
  <owl:Class rdf:about="#Person"/>
 </rdfs:range>
 <rdf:type>
   <owl:SymmetricProperty/>
 </rdf:type>
</owl:ObjectProperty>

<rdf:Description rdf:about="#Bill">
  <friend rdf:resource="#Ted"/>
</rdf:Description>
```

The friend property is symmetric (see 6.6.7), so we know that as well as Ted being a friend of Bill, Bill will be a friend of Ted. The hasValue restriction (see 6.3.10) can now be used to talk about the friends of the particular individual Bill (as a class):

```
EquivalentClasses (
 FriendOfBill
 restriction (friend value=Bill)
)
```

```
<owl:Class rdf:ID="FriendOfBill">
 <owl:sameClassAs>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#friend"/>
   <owl:hasValue rdf:resource="#Bill"/>
  </owl:Restriction>
 </owl:sameClassAs>
</owl:Class>
```

7.7 Inconsistency and Unsatisfiability

As we have seen above, the axioms and assertions in an ontology can interact and allow us to draw inferences about the subclass relationships that exist between classes. In addition, we can discover *inconsistencies*. An inconsistent (or *unsatisfiable*) class is one for which we can deduce that it is impossible for the class ever to have any instances. As an example, the class Impossible:

```
EquivalentClasses (
 Impossible
 intersectionOf (Man complementOf (Man))
)
```

```
<owl:Class rdf:ID="Impossible>
 <owl:subClassOf>
  <owl:Class>
    <owl:intersectionOf rdf:parseType="owl:collection">
      <owl:Class rdf:about="#Man"/>
      <owl:complementOf>
       <owl:Class rdf:about="#Man"/>
      </owl:complementOf>
    </owl:intersectionOf>
  </owl:Class>
 </owl:subClassOf>
</owl:Class>
```

is inconsistent. In order for there to be an instance a of Impossible, it must be both an instance of the class Man and not an instance of the class Man.

As ever, we must be careful about inconsistency and unsatisfiability. For example, we may introduce a class Unicorn:

```
SubClassOf (Unicorn Animal)
```

```
<owl:Class rdf:ID="#Unicorn">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#Animal"/>
 </rdfs:subClassOf>
</owl:Class>
```

Our knowledge of the world tells us that there are actually no Unicorns that exist, but this does not mean that this particular class is unsatisfiable. There is nothing here in the intensional definitions of the ontology that allows us to logically deduce that there *cannot* be any Unicorns (which is a stronger assertion than the fact there are none).

The circumstances that create inconsistent or unsatisfiable classes can be much more complicated than the (somewhat trivial) Impossible example shown above. For example, consider the case where the following axiom has been added:

```
SubClassOf (
  restriction (owns minCardinality(4))
  restriction (hasPet someValuesFrom = Dog)
)
```

```
<owl:Restriction>
 <owl:onProperty rdf:resource="#owns"/>
 <owl:minCardinality>4</owl:minCardinality>
 <rdfs:subClassOf>
   <owl:Restriction>
    <owl:onProperty rdf:resource="#hasPet"/>
    <owl:someValuesFrom>
     <owl:Class rdf:about="#Dog"/>
    </owl:someValuesFrom>
   </owl:Restriction>
  </rdfs:subClassOf>
</owl:Restriction>
```

This axiom tells us that anything that owns more than 4 pets, must have a Dog as a pet[19]. If we now consider the following class describing Old Ladies with lots of pets[20]:

```
EquivalentClasses (
  OldLadyWithLotsofPets
  intersectionOf (
    OldLady
    restriction (hasPet minCardinality = 5)
  )
)
```

```
<owl:Class rdf:about="#OldLadyWithLotsOfPets">
 <owl:sameClassAs>
  <rdfs:Class>
   <owl:intersectionOf rdf:parseType="owl:collection">
    <owl:Class rdf:about="#OldLady"/>
    <owl:Restriction>
     <owl:onProperty rdf:resource="#hasPet"/>
```

[19] Again, no claim is being made as to the ontological accuracy of this sample model. One might claim that the ontology being produced here is inconsistent with our observed view of the world – this, however is a different notion of inconsistency to that of *formal* or *logical* inconsistency as is being discussed here.

[20] The lengthy class names being introduced here serve to illustrate a useful feature of languages like OWL. In the past, thesauri and controlled vocabularies tended to rely on the semantics of the terms being provided in the main by the *names* of the classes. So the term OldLadyWithLotsOfPets represented the class of Old Ladies with lots of pets, but it would not necessarily be clear exactly what "lots of pets" meant in this context – 3?, 10?, 579? A *scope note* or documentation may assist in deciding on the intended interpretation of the term, but such information is not always amenable to computation, or even accessible to machines. With the OWL definition, we have been explicit about what our intended interpretation of "lots of pets" is here – it is at least 5.

```
     <owl:minCardinality>5</owl:minCardinality>
    </owl:Restriction>
   </owl:intersectionOf>
  </rdfs:Class>
 </owl:sameClassAs>
</owl:Class>
```

we can deduce that this class is unsatisfiable – i.e. there can never be an instance of it. Why is this? The axiom introduced above tells us that whenever an individual owns more than 4 things, it must have a Dog as a pet. Instances of the class OldLadyWithLotsOfPets must have at least 5 pets. So they must own at least 5 things (as hasPet is a subproperty of owns), thus own at least 4, and hence must have a Dog as a pet. But there is also the axiom introduced earlier that tells us that the *only* pets that an OldLady can have are Cats, and an additional axiom that tells us that Cats and Dogs are disjoint. Thus we have arrived at a contradiction, and it is impossible for any individual to satisfy the conditions for membership of the class OldLadyWithLotsOfPets. It is easy to see how this kind of deduction can quickly become a difficult and complex process, particularly in ontologies with large numbers of axioms or constraints. This is one example of where the use of a reasoner can help, assisting in checking OWL ontologies for consistency.

Acknowledgments

The authors would like to acknowledge the contribution of all those involved in the development of DAML-ONT, OIL and DAML+OIL, amongst whom Dieter Fensel, Frank van Harmelen, Deborah McGuinness and Peter F. Patel-Schneider deserve particular mention.

References

[1] Sean Bechhofer, Carole Goble, and Ian Horrocks. DAML+OIL is not enough. In *Proc. of the 2001 Int. Semantic Web Working Symposium (SWWS 2001)*, pages 151–159, 2001. Available at http://www.semanticweb.org/SWWS/program/full/SWWSProceedings.pdf.

[2] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: A Reasonable ontology editor for the semantic web. In *Proc. of the Joint German/Austrian Conf. on Artificial Intelligence (KI 2001)*, number 2174 in Lecture Notes in Artificial Intelligence, pages 396–408. Springer, 2001. Appeared also in Proc. of the 2001 Description Logic Workshop (DL 2001).

[3] Tim Berners-Lee. *Weaving the Web*. Harpur, San Francisco, 1999.

[4] Alexander Borgida and Peter F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *J. of Artificial Intelligence Research*, 1:277–308, 1994.

[5] S. Decker, F. van Harmelen, J. Broekstra, M. Erdmann, D. Fensel, I. Horrocks, M. Klein, and S. Melnik. The semantic web: The roles of XML and RDF. *IEEE Internet Computing*, 4(5), 2000.

[6] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In R. Dieng, editor, *Proc. of the 12th European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW'00)*, number 1937 in Lecture Notes in Artificial Intelligence, pages 1–16. Springer-Verlag, 2000.

[7] Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[8] Richard E. Fikes and Deborah L. McGuinness. An axiomatic semantics for RDF, RDF Schema, and DAML+OIL. Technical Report KSL-01-01, Stanford University KSL, 2001. Available at `http://www.ksl.stanford.edu/people/dlm/daml-semantics/abstract-axiomati%c-semantics.html`.

[9] Erich Grädel, Martin Otto, and Eric Rosen. Two-variable logic with counting is decidable. In *Proc. of the 12th IEEE Symp. on Logic in Computer Science (LICS'97)*, pages 306–317. IEEE Computer Society Press, 1997.

[10] W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, and M. A. Musen. Knowledge modelling at the millenium (The design and evolution of Protégé-2000). In *Proc. of Knowledge acqusition workshop (KAW'99)*, 1999.

[11] James Hendler and Deborah L. McGuinness. The darpa agent markup language". *IEEE Intelligent Systems*, 15(6):67–73, 2000.

[12] Bernhard Hollunder and Franz Baader. Qualifying number restrictions in concept languages. In *Proc. of the 2nd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'91)*, pages 335–346, 1991.

[13] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer-Verlag, 1999.

[14] Ian Horrocks. Reasoning with expressive description logics: Theory and practice. In Andrei Voronkov, editor, *Proc. of the 18th Int. Conf. on Automated Deduction (CADE 2002)*, number 2392 in Lecture Notes in Artificial Intelligence, pages 1–15. Springer, 2002.

[15] Ian Horrocks and Ulrike Sattler. Ontology reasoning in the $\mathcal{SHOQ}$(D) description logic. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, pages 199–204, 2001.

[16] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer, 1999.

[17] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for very expressive description logics. *J. of the Interest Group in Pure and Applied Logic*, 8(3):239–264, 2000.

[18] D. L. McGuinness. Ontological issues for knowledge-enhanced search. In *Proc. of FOIS*, Frontiers in Artificial Intelligence and Applications. IOS-press, 1998.

[19] D. L. McGuinness. Ontologies for electronic commerce. In *Proc. of the AAAI '99 Artificial Intelligence for Electronic Commerce Workshop*, 1999.

[20] S. McIlraith, T.C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, March/April 2001.

[21] Stephan Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen, 2001.