Report on Semantic Web Languages Evolution

Uwe Keller, Anna V. Zhdanova, Sinuhé Arroyo
Institute for Computer Science
University of Innsbruck

17-01-2004
Executive Summary

This deliverable covers two objectives. First, it delineates the changes and evolution that have taken place in the Semantic Web Languages field since the last deliverable (D2.1), regarding the most important ontology languages that were introduced in D2.1, such as RDF/S, OIL, DAML+OIL, OWL. Second, this deliverable is intended to help in the decision making process of which Semantic Language to use in the Esperonto project.

During the run-time of the Esperonto project, the objective of the deliverable 2.2 is to provide an up-to-date information about evolution of the Semantic Web languages and technologies supporting these languages.
## Document Information

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>IST-2001-34373</th>
<th>Acronym</th>
<th>Esperonto Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full title</td>
<td>Application Service Provision of Semantic Annotation, Aggregation, Indexing and Routing of Textual, Multimedia, and Multilingual Web Content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project URL</td>
<td><a href="http://www.esperonto.net">www.esperonto.net</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document URL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EU Project officer</td>
<td>Werner Janusch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>Name</th>
<th>Report on Semantic Web languages evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Number</td>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>Work package Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date of delivery</td>
<td>Contractual</td>
<td>31-08-2004</td>
</tr>
<tr>
<td>Code name</td>
<td>Status</td>
<td>draft; final</td>
</tr>
<tr>
<td>Nature</td>
<td>Prototype; Report; Specification; Tool; Other</td>
<td></td>
</tr>
<tr>
<td>Distribution Type</td>
<td>Public; Restricted; Consortium</td>
<td></td>
</tr>
<tr>
<td>Authors (Partner)</td>
<td>Uwe Keller (IFI), Anna V. Zhdanova (IFI), Sinuhé Arroyo (IFI)</td>
<td></td>
</tr>
<tr>
<td>Contact Person</td>
<td>Anna V. Zhdanova</td>
<td></td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:anna.zhdanova@uib.k.ac.at">anna.zhdanova@uib.k.ac.at</a></td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>+43 512 507 6467</td>
<td></td>
</tr>
<tr>
<td>Fax</td>
<td>+43 512 507 9872</td>
<td></td>
</tr>
</tbody>
</table>

**Abstract**

This deliverable describes the changes and evolution of the Semantic Web languages that happened since the time of the issue of the first version of this deliverable.

**Keywords**

OWL, RDF, RDFS Semantic Languages, OIL, DAML+OIL, WSMO

<table>
<thead>
<tr>
<th>Version log/Date</th>
<th>Change</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>11-08-2003</td>
<td>First version</td>
<td>Sinuhé Arroyo</td>
</tr>
<tr>
<td>18-08-2003</td>
<td>First version – second revision</td>
<td>Sinuhé Arroyo</td>
</tr>
<tr>
<td>26-08-2003</td>
<td>QA</td>
<td>Richard Benjamins</td>
</tr>
<tr>
<td>28-08-2003</td>
<td>First version – final</td>
<td>Sinuhé Arroyo</td>
</tr>
<tr>
<td>01-07-2004</td>
<td>Second version</td>
<td>Anna V. Zhdanova</td>
</tr>
<tr>
<td>07-08-2004</td>
<td>Second version</td>
<td>Uwe Keller</td>
</tr>
<tr>
<td>08-08-2004</td>
<td>QA</td>
<td>Oscar Corcho</td>
</tr>
<tr>
<td>24-08-2004</td>
<td>Second version - final</td>
<td>Anna V. Zhdanova and Uwe Keller</td>
</tr>
<tr>
<td>17-01-2005</td>
<td>Third version</td>
<td>Uwe Keller</td>
</tr>
</tbody>
</table>
## Project Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| Intelligent Software Components S.A. (Coordinator) | iSOCO | Dr. V. Richard Benjamins  
c/ Pedro de Valdivia, 10  
28006 Madrid, Spain  
#e richard@isoco.com  
#t +34-91-334-97-97, #f +34-91-334-97-99 |
| Universidad Politécnica de Madrid | UPM | Dr. Asunción Gómez-Pérez  
Campus de Montegancedo, sn  
Boadilla del Monte, 28660, Spain  
#e asun@it.upm.es  
#t +34-91 336-7439, #f +34-91 352-4819 |
| Institut für Informatik, Leopold-Franzens Universität Innsbruck | IFI | Prof. Dieter Fensel  
Institute of computer science  
University of Innsbruck  
Technikerstr. 25  
A-6020 Innsbruck, Austria  
#e dieter.fensel@uibk.ac.at  
#t +43 512 507 6486 |
| Universität des Saarlandes | UdS | Thierry Declerck  
DFKI GmbH (German Research Center for AI),  
Stuhlsatzenhausweg 3, D-66123 Saarbruecken (Germany)  
#e: declerck@dfki.de  
#t: +49-681-302-5358, #f: +49-681-302-5338 |
| The University of Liverpool | UniLiv | Dr. Valentina A.M. Tamma  
Department of Computer Science,  
University of Liverpool  
Room 1.11, Chadwick Building  
Peach Street  
Liverpool L69 7ZF, UK  
#e valli@csc.liv.ac.uk  
#t +44 151 794 6797, #f +44 151 794 3715 |
| Fundación Residencia de Estudiantes | Residencia | Elisa Navas  
Fundación Residencia de Estudiantes  
Pinar, 23  
28006 Madrid, Spain  
#e enavas@fundacionqiner.org  
#t +34-91-446 01 97, #f +34-91-4468068 |
| Centré d'Innovació i Desenvolupament Empresarial | CIDEM | Carlos Gómar  
Centré d'Innovació i Desenvolupament Empresarial  
Provença, 339  
08037 Barcelona, Spain  
#e cgomara@cidem.gencat.es  
#t +34-93-4767305, #f +34-93-4767303 |
| Biovista | Biovista | Dr. Andreas Persidis  
34 Rodopoleos Street  
Ellinikon  
Athens 16777, HELLAS  
#e biovista@ath.forthnet.gr  
#t +30.1.9629848, #f +30.1.9647606 |
# Table of Contents

1. Introduction ................................................................. 5

2. Semantic Web Languages Evolution .................................. 6
   2.1 RDF Schema ................................................................ 6
      2.1.1 Overview ......................................................... 6
      2.1.2 Updates and improvements: October 2002-August 2003 .......... 6
      2.1.3 Updates and improvements: August 2003-August 2004 ............. 7
      2.1.4 Summary ....................................................... 7
   2.2 OIL ........................................................................... 8
      2.2.1 Overview ......................................................... 8
      2.2.2 Updates and improvements: October 2002-August 2003 .......... 9
      2.2.3 Updates and improvements: August 2003-August 2004 ............. 9
      2.2.4 Summary ....................................................... 9
   2.3 DAML+OIL ............................................................... 9
      2.3.1 Overview ......................................................... 9
      2.3.2 Updates and improvements: October 2002-August 2003 .......... 9
      2.3.3 Updates and improvements: August 2003-August 2004 ............. 10
      2.3.4 Summary ....................................................... 10
   2.4 OWL ........................................................................... 10
      2.4.1 Overview ......................................................... 10
      2.4.2 Updates and improvements: October 2002-August 2003 .......... 11
      2.4.3 Updates and improvements: August 2003-August 2004 ............. 13
      2.4.4 Summary ....................................................... 13
   2.5 Ontology languages in the WSMO project ......................... 14
      2.5.3 Summary ....................................................... 33
   2.6 Ontology language in the DIP project .............................. 63
      2.6.1 Overview ......................................................... 63
      2.6.3 Summary ....................................................... 63
   3. Choosing an Ontology Language ...................................... 65
   3.1 Requirements of the Esperonto project ............................. 65
   3.2 Candidate languages .................................................. 66
      3.2.1 RDF(s) ............................................................ 66
      3.2.2 OIL ................................................................. 66
      3.2.3 DAML+OIL .................................................... 66
      3.2.4 OWL ............................................................... 67
      3.2.5 Ontology languages in the WSMO initiative ......................... 67
      3.2.6 Ontology language of the DIP project ............................. 68
   3.3 Rule extensions of candidate languages ............................ 68
      3.3.1 RDF/S rule extensions ....................................... 70
      3.3.2 OWL rule extensions ....................................... 73
   3.4 Candidate solution .................................................... 76

4. Conclusions and Future Plan ............................................ 77

5. References ....................................................................... 78
1. Introduction

The current World Wide Web (WWW) is, by its function, the syntactic web where structure of the content has been presented while the content itself is difficult to access to computers. Although the WWW has resulted in a revolution in information exchange among computer applications, it still cannot fulfil the interoperation among various applications without some pre-existing, human-created agreements somewhere in-house or outside of the web.

The next generation of the Web aims to alleviate such problem. The Web resources will be much easier and more readily accessible by both human and computers with the added semantic information in a machine-understandable and machine-processible fashion [Berners-Lee99]. "The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation," [Berners-Lee01].

The practical problem is how to make the Semantic Web come true, i.e., make possible for a computer to interpret the semantic meaning of the information presented on the Web. Ontologies here are the silver-bullet. They play a key role by providing shared and precisely defined terms that can be understood and processed by machines. A typical ontology consists of a hierarchical description of important concepts and their relations in a domain, task or service. 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. Therefore an effective ontology language which can help in formalization of the web is the most wanted thing on the Semantic Web. D1.1 can be referred for further information.

As Tim Berners-Lee described, the XML/RDF/OWL family Semantic Web languages can be structured in a layered manner. The layered tower is the dreamed vehicle to bring the Web to its full potential. The recognition of the importance of ontologies for the Semantic Web has led to the revolution and extension of the current web markup languages, e.g., XML Schema, RDF (Resource Description Framework), and RDF Schema, furthermore, OIL, DAML+OIL, and OWL.

In this deliverable, we intend to provide up-to-date information about the most relevant existing web ontology languages, such as RDF(s), OIL, DAML+OIL, and OWL, as well as help in the decision which of these languages should be used in the Esperonto project. The structure of this survey is planned as follows: In Section 2 – Semantic Web Languages Evolution, we give a short introduction to each of the languages, then the main updates and improvements are presented, and finally a summary of the information provided is facilitated. In Section 3, the aim is to help to decide which of all the available semantic languages is the best choice for the Esperonto project. Section 4 contains the final summary.
2. Semantic Web Languages Evolution

In this section an overview of the improvements and updates on the most relevant Semantic Web languages is presented. The process starts with RDF(s), continues with OIL, and DAML+OIL, and finishes with OWL. For each of these languages a short description is provided, then, the main updates and improvements for the first and the second versions of this deliverable, and finally, a short summary of the information provided.

The deliverable deals with evolution of Semantic Web languages. Our understanding of evolution is close to the one that has originated and is commonly used in natural sciences. can be expressed in the following definition adapted from Darwin [Darwin].

**Definition:** evolution is a progress over time in variety and adaptability of the object’s properties.

Obviously, with respect to this definition, examination of Semantic Web language properties requires explicit attention in this deliverable. Specifically, the important issues to take into account are:

- how the language properties change
- how the language properties “adapt” to the changing outside world.

### 2.1 RDF Schema

This part presents a short overview of RDF Schema, followed by a summary of the main updates and improvements that the languages has suffered in the last months, and finished with a summary of the information presented.

#### 2.1.1 Overview

In the Semantic Web, people are capable to describe the attributes of certain kinds of resources, for instance, to describe the "author", "title", and "subject" for certain bibliographic resources. The declaration of these properties (attributes) and their corresponding semantics are defined in the context of RDF as a RDF schema\(^1\). A schema defines not only the properties of the resource (e.g., title, author, subject, size, colour, etc.) but also the kinds of resources being described (books, Web pages, people, companies, etc.).

#### 2.1.2 Updates and improvements: October 2002-August 2003

The RDF Core working group has focused in the following efforts:

- Revise the RDF Model and Syntax Recommendation.
- Complete work on the RDF Schema specification and provide a means to support tighter integration with the XML Schema Part 2: Data types Recommendation

As an output of this work the following Last Call Working Drafts have been produced:

- **RDF/XML Syntax Specification (Revised)**\(^2\), Updates the grammar in the RDF Model and Syntax Recommendation and addresses questions that have been raised about parts of the RDF 1.0 specification.

---

1. [http://www.w3.org/TR/rdf-schema/](http://www.w3.org/TR/rdf-schema/)
• **Resource Description Framework (RDF) Concepts and Abstract Syntax**\(^3\). Defines the abstract graph syntax on which RDF is based, and serves to link its XML serialization to its formal semantics.

• **RDF Vocabulary Description Language 1.0**\(^4\). RDF Schema describes how to use RDF to describe RDF vocabularies.

• **RDF Primer**\(^5\). Provides a tutorial on the fundamentals required to use RDF in applications.

• **RDF Semantics**\(^6\). Specifies a precise semantic theory for RDF Model and Syntax and RDF Schema, and of corresponding entailment and inference rules which are sanctioned by the semantics.

• **RDF Test Cases**\(^7\). Provides a set of machine processable test cases corresponding to technical issues addressed by the Working Group.

### 2.1.3 Updates and improvements: August 2003-August 2004

**RDF/S content change**

- rdf:RDF made optional The rdf:RDF element was made optional when there is only one outer node element (inside rdf:RDF) in an RDF/XML document.
- Unicode Normal Form C (NFC) checks Modified the NFC checks to be optional (SHOULD) rather than required (MUST) and added some missing checks.
- Lexical and value spaces of datatypes are required to be nonempty
- "Vocabulary" is now defined to include typed literals as well as URI references; definition of 'name' changed accordingly
- The set LV of literal values is no longer considered 'global' but is part of an interpretation
- RDF lists are no longer required to explicitly give rdf:type rdf:List triples for all sublists
- The reference [RDF_VOCABULARY] is not a normative reference

**RDF/S status change**

- **W3C Working Draft 05 September 2003**
- **W3C Working Draft 10 October 2003**
- **W3C Proposed Recommendation 15 December 2003**
- **W3C Recommendation 10 February 2004**

### 2.1.4 Summary

As stated in this section RDFs can be envisioned as the language to define the container of the information that wants to be stored. It describes the properties of the resources as well as the resources itself.

The main updates and improvements accomplished within the context of this language are related to grammar updates, ways to link the XML serialization to formal semantics, tutorials on how to use RDF in applications, syntax and test cases.

---

3. [http://www.w3.org/TR/rdf-concepts/](http://www.w3.org/TR/rdf-concepts/)
7. [http://www.w3.org/TR/rdf-testcases/](http://www.w3.org/TR/rdf-testcases/)
2.2 OIL

This part presents a short overview of OIL, followed by a summary of the main updates and improvements that the languages has undergone in the last months, and is with a summary of the information presented.

2.2.1 Overview

OIL\(^8\) is built on top of RDF and RDFS, using as much as possible their constructs in order to maintain backward compatibility. OIL provides modelling primitives used in frame-based and Description Logic oriented ontologies, coming along with a simple and clean semantics. It has a syntax definition using web standards such as RDF(s) and XML(s).

OIL unifies three important aspects provided by different communities: (1) formal semantics and efficient reasoning support as provided by Description Logic, (2) epistemologically rich modelling primitives as provided by the Frame-based community, and (3) a standard proposal for syntactical exchange notations as provided by the Web community.

OIL presents a layered architecture formed by:

- **Core OIL** coincides largely with RDF Schema (with the exception of the reification features of RDF Schema).
- **Standard OIL** is a language intended to capture the necessary main stream modelling primitives that both provide adequate expressive power and are well understood by allowing the semantics to be precisely specified and complete inference.
- **Instance OIL** includes a thorough individual integration. While the previous layer - Standard OIL - included modelling constructs that allow individual fillers to be specified in term definitions, Instance OIL includes a full-fledged database capability.
- **Heavy OIL** may include additional representational (and reasoning) capabilities.

Such a layered organization has three main advantages: First, an application is not forced to work with a language that offers significant more expressiveness and complexity than it actually needs. Second, applications that can only process a lower level of complexity are still able to catch some of the aspects of an ontology. Third, an application that is aware of a higher level of complexity can still also understand ontologies expressed in a simpler ontology language.

\(^8\) [http://www.ontoknowledge.org/oil/](http://www.ontoknowledge.org/oil/)
2.2.2 Updates and improvements: October 2002-August 2003

There are neither updates nor improvements in this language since its development stopped some time ago. The natural continuer of the work carried here is DAML+OIL a joint effort of the American and European ontology communities for the Semantic Web. Anyhow the latest updates that can be found in the OIL site refers to extensions of the core language with additional primitives for which there will be no reasoning support, and the release of version 3.5 of the editor OilEd⁹.

2.2.3 Updates and improvements: August 2003-August 2004

OIL language is not noticed to be developed further.

2.2.4 Summary

OIL is the next natural step in the Semantic Web languages development process, right after RDF. It is built on top of RDF and RDFs, and adds a unification of important aspects pointed by different communities such as formal semantics, description logics and a standard proposal for syntactical exchange notations. It has a layered architecture that provides a great degree of flexibility.

Regarding the updates and improvements, must be stated that there has been none from D2.1 was made available, since this is a dead language which is no longer under development.

Resources

http://www.ontoknowledge.org/oil/

2.3 DAML+OIL

This part presents a short overview of DAML+OIL, followed by a summary of the main updates and improvements that the languages has suffered in the last months, and finished with a summary of the information presented.

2.3.1 Overview

DAML+OIL¹⁰ is an ontology language specifically designed for the Semantic Web, created as a joint effort of the American and European ontology communities for the Semantic Web. It exploits existing Web standards (XML and RDF), adding the ontological primitives of object oriented and frame-based systems, and the formal rigor of expressive description logic. As an ontology language, DAML+OIL is designed to describe the structure of a domain. DAML+OIL takes an object-oriented approach, with the structure of the domain being described in terms of classes and properties, and the set of axioms that assert characteristics of these classes and properties.

2.3.2 Updates and improvements: October 2002-August 2003

---

⁹ http://oiled.man.ac.uk/
¹⁰ http://www.daml.org/
Since December 2001 there has been no work in order to improve or update DAML+OIL. The last working drafts submitted to the W3C consortium are:

- **A Model-Theoretic Semantics for DAML+OIL (March 2001)**. 18 December 2001, Frank van Harmelen, Ian Horrocks, Peter F. Patel-Schneider (DAML+OIL Web Ontology Language Submission)
- **DAML+OIL (March 2001) Reference Description**. 18 December 2001, Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein (DAML+OIL Web Ontology Language Submission)
- **Annotated DAML+OIL Ontology Markup**. 18 December 2001, Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein (DAML+OIL Web Ontology Language Submission)

Apart from this no much work has been carried on in this language. Notice that all this updates and improvements are previous to the publication of D2.1.

### 2.3.3 Updates and improvements: August 2003-August 2004

DAML+OIL language is not noticed to be developed further.

### 2.3.4 Summary

DAML+OIL represent a joint effort of Ontology communities in Europe (OIL) and in the American to align their efforts. Following the language layer sketched by Tim Berners-Lee it is built on top RDF and XML with improvements coming from the frame based systems and description logics. The development of this language was stopped as far as the W3C is concerned in 2001 when the last working drafts were submitted.

**Resources**

http://www.daml.org/2001/03/daml+oil-index.html
http://www.w3.org/TR/daml+oil-reference

### 2.4 OWL

This part presents a short overview of OWL, followed by a summary of the main updates and improvements that the languages has suffered in the last months, and finished with a summary of the information presented.

#### 2.4.1 Overview

OWL\(^{11}\) is the web ontology language currently under the development of W3C Web Ontology (WebOnt\(^{12}\)) Working Group. OWL is mainly based on OIL and DAML+OIL and therefore the main features of OWL are very similar to those of OIL. OWL includes three sub languages called:

- **OWL-Lite**. Roughly consists of RDFS plus equality and 0/1-cardinality. It represents a migration path from other taxonomies. It is intended for classification hierarchies and

---

\(^{11}\) [http://www.w3.org/TR/owl-ref/](http://www.w3.org/TR/owl-ref/)

\(^{12}\) [http://www.w3.org/2001/sw/WebOnt/](http://www.w3.org/2001/sw/WebOnt/)
simple constraints. It should be kept as simple as possible in order to facilitate the tool development.

- **OWL DL.** Contains the language constructs but with hierarchy restrictions. It provides computational completeness and decidability, and it is dotted with the maximum expressive power.

- **OWL Full.** Composed by the complete vocabulary interpreted more broadly than in OWL DL. It incorporates maximum expressive power and syntactic freedom. It offers no computational guarantees. It is not very likely to happen that any reasoning engine will support the reasoning capabilities of every OWL feature.

Besides the DAML+OIL style RDF syntax, the OWL specification also includes an abstract syntax, which provides a higher level and less cumbersome way of writing ontologies.

The OWL language can be used to allow the explicit representation of term vocabularies and the relationships between entities. The OWL language is a revision of the DAML+OIL web ontology language-incorporating lesson learned from the design and application use of DAML+OIL.

Main features of OWL language include:

- **Ontologies.** OWL ontology is a sequence of axioms and facts, plus inclusion references to other ontologies, which are considered to be included in the ontology. OWL ontologies are web documents, and can be referenced by means of a URI. Ontologies also have a non-logical component (not yet specified) that can be used to record authorship, and other non-logical information to be associated with a ontology.

- **Axioms.** Axioms are used to associate class and property IDs with either partial or complete specifications of their characteristics, and to give other logical information about classes and properties. It contains Class Axioms, Property axioms, Descriptions and Restrictions.

- **Facts.** Facts state information about particular individuals in the form of a class that the individual belongs to plus properties and values. Individuals can either be given an individualID or be anonymous (blank nodes in RDF terms). The syntax here is set up to mirror the normal RDF/XML syntax.

### 2.4.2 Updates and improvements: October 2002-August 2003

The most recently publish documents about OWL are the “OWL Last Call Working Drafts” which were made available on 2003-04-02. In this working draft the following improvements should be highlighted:

- More expressive power has been added to sublanguages (OWL-Lite, OWL DL and OWL Full)

- Support for ontology mapping has been included. The primitives that support this functionality are: `equivalentClass`, `equivalentProperty`, `sameIndividualAs`, `differentFrom`, `allDifferent`.

- Mapping to RDF Graphs. It defines a many-to-many relationship between abstract syntax ontologies and RDF graphs. This is done using a set of nondeterministic mapping rules. It allows:

  - Translation to RDF graphs
  - Definition of OWL DL and OWL-Lite in graph form

The list of construct of the language has been improved, in this part of the document an incremental enumeration of such constructs is presented for each of the OWL languages. For
more information on the different language constructs the reader can take a look at the “OWL Web Ontology Language Overview”\(^{13}\).

The list of updated language constructs for OWL-Lite is as follows.

\(^{13}\) http://www.w3.org/TR/owl-features/
The inclusion of \((\text{In})\text{Equality}\) constructs is of special importance for the Esperonto project, since it is relevant for the Esperonto Ontology Alignment Solution (D1.4) [deBruijn03] in the case OWL is chosen as a preferred language for the project.

It should be noted that many of the OWL properties including the ones listed here have restrictions on their application, e.g. they can only be applied to named classes. Hence there can be a problem with expressing arbitrary complex expressions.

The list of updated language constructs for OWL DL and OWL Full is as follows.

**Class Axioms:**
- oneOf
- disjointWith
- equivalentClass
- rdfs:subClassOf

**Arbitrary Cardinality:**
- minCardinality
- maxCardinality
- cardinality

**Filler Information:**
- hasValue

**Boolean Combinations of Class Expressions:**
- unionOf
- intersectionOf
- complementOf
The **Web Ontology** working group has focused its efforts on extending OWL definition producing the following candidate recommendations:

- **OWL Web Ontology Language Overview**\(^{14}\). Provides an introduction to OWL and to a subset called OWL Lite.
- **OWL Web Ontology Language Guide**\(^{15}\). Demonstrates the use of the OWL language to formalize a domain.
- **OWL Web Ontology Language Test Cases**\(^{16}\). Presents test cases for the Web Ontology Language (OWL).
- **OWL Web Ontology Language Semantics and Abstract Syntax**\(^{17}\). Provides a high-level, abstract syntax for both OWL and OWL Lite, a subset of OWL.
- **OWL Web Ontology Language Reference**\(^{18}\). Provides a systematic, compact and informal description of all the modeling primitives of OWL.
- **OWL Web Ontology Language Use Cases and Requirements**\(^{19}\). Illustrates correct OWL usage.

### 2.4.3 Updates and improvements: August 2003-August 2004

**OWL content change**

The most significant update in OWL was introduction of the `owl:Nothing` construction that represents an empty class. This construction was added to OWL Lite.

After OWL has gained recognition to be on its steady way to becoming eventually a W3C recommendation, a major part of efforts of the WebOnt working group were connected with the quality of presentation of W3C OWL documents. In particular, the improvements are significantly connected with making examples more understandable and correcting typos.

**OWL status change**

- **W3C Candidate Recommendation 18 August 2003**
- **W3C Proposed Recommendation 15 December 2003**
- **W3C Recommendation 10 February 2004**

### 2.4.4 Summary

OWL represent the next generation of Semantic Web languages still under development of W3C Web Ontology Working Group. OWL is inspired by OIL and DAML+OIL, and it is structured in three sublanguages in order to cover a broad band of necessities. The latest main improvements in the language are related to the power expressiveness and the development of new constructs. The Web Ontology working group has been very active delivering a large number of last call working drafts, among which it is especially important to

---


highlight the “OWL Web Ontology Language Semantics and Abstract Syntax” that condenses the main improvements. At the moment, OWL is at the pruning stage, and frequent and major changes are not expected to take place. The main efforts of the Web Ontology Working Group were switched to the activities within the newly established W3C Semantic Web Best Practices and Deployment Working Group (SWBPD)\(^\text{20}\). The major efforts of the SWBPD group aim at providing hands-on support for developers of Semantic Web applications, which will contribute to dissemination and pruning of OWL (and RDF) languages.

**Resources**

http://www.w3.org/TR/owl-features/
http://www.w3.org/TR/owl-guide/
http://www.w3.org/TR/owl-test/
http://www.w3.org/TR/webont-req/
http://www.w3.org/TR/owl-ref/
http://www.w3.org/TR/owl-semantics/

### 2.5 Ontology languages in the WSMO project

**The WSMO project.** The Web Service Modelling Ontology (WSMO)\(^\text{21}\) project is the major European initiative in Semantic Description of Web Services. It is carried out by the SDK-Project Cluster\(^\text{22}\) in the context of three EU-funded projects, namely SEKT\(^\text{23}\), DIP\(^\text{24}\) and Knowledge Web\(^\text{25}\), and aims at providing the conceptual model for semantically describing various aspects of Web Services which are relevant for discovery, composition and mediation.

According to the mission statement of the project, the main features of WSMO – simplicity (a solution to the integration problem that is as simple as possible), completeness (solves all aspects of the integration problem), executability (a set of execution semantics exists as well as a reference implementation) - should provide a world-wide standard, which will be developed together with industrial partners and other research groups, and will be aligned with many different research projects.

The pillars of the project are provided by the Web Service Modeling Framework (WSMF), which serves as a conceptual basis that will be further developed in the course of the project. Moreover, the project aims at alignment with current available initiatives and standards that try to address similar problems and overcome drawbacks in existing approaches.

The WSMO initiative hosts two sub-projects for modelling language related issues (Web Service Modeling Language, WSML) and for the design and implementation of a reference implementation (Web Service Execution Environment, WSMX).

In the context of the WSML project there are some efforts related to developing ontology languages with specific characteristics that particularly useful in the area of Semantic Web Services, namely OWL-Lite-, OWL-Flight, OWL-DL-, OWL-Full-, WSML-Core. Although, these languages are being developed in a specific context, namely Semantic Web Service...
Description, they are useful and relevant in their own right. We will briefly discuss these languages in the following sections.

a. OWL-Lite-

OWL Lite is the least expressive species of OWL. However, this language already requires reasoning with equality, which significantly increases computational complexity. Cardinality restrictions, in their current form, introduce equality in a non-intuitive way. There is no notion of constraints in OWL Lite. Furthermore, because the expressiveness of the SHIF Description Logic language is beyond the capabilities of efficient rule-based engines, and because straightforwardly extending a Description Logic with Horn-like rules leads to undecidability issues [Levy and Rousset, 1998], one cannot easily extend OWL Lite with a rule language without loosing computational guarantees which so far has been considered as an important feature of Semantic Web languages.

[de Bruijn, 2004] defines the ontology language OWL Lite-, which is a proper subset of OWL Lite that can be translated into Datalog. OWL-Lite- restricts the syntax and semantics of OWL Lite.

The authors argue that some of the above mentioned limitations can be overcome by using a more restricted form of OWL Lite, which can be translated into a Datalog program (without equality). This language can then be straightforwardly extended to include database-style integrity constraints, which can be used for both cardinality and value constraints. Furthermore, in Datalog rules can be added directly on top of the ontology.

The invention of OWL Lite- is based on results of the PhD thesis by Raphael Volz [Volz, 2004], namely the language L₀ defined in the thesis. The motivation and justification for the specific construction of OWL Lite and L₀ and can be summarized as follows:

- There exist many efficient implementations of Datalog, which are very efficient at the task of query answering.
- It turns out that most ontologies currently on the Semantic Web can be expressed in the language L₀ [Volz, 2004].
- It turns out that nearly all of L₀ is included in OWL Lite. In fact, it turns out that the only construct which is in L₀, but is not in OWL Lite, is the hasValue property restriction $\exists R\{o\}$.

Table 1 [de Bruijn, 2004] overviews the OWL Lite- language constructs and the elements of OWL Lite which are not present in OWL Lite.

Roughly speaking, one can summarize that all elements of OWL Lite have been removed which add cardinality constraints or implicit equality statements. Furthermore, owl:Thing and owl:Nothing are not directly representable in OWL Lite-, whereas the authors mention that in principle there would be no problem with allowing these constructs in OWL Lite-; in this case, the use of both concepts would be restricted in class definitions as well as domain and range restrictions.
Because OWL Lite- is a proper subset of OWL Lite, it (almost) automatically comes along with an RDF syntax and a model-theoretic semantics.

The relation of OWL-Lite- and RDFS can be summarized as follows: OWL Lite is (partly) syntactically and (partly) semantically layered on top of RDFS. There is, however, a subset of RDFS which is both syntactically and semantically included in OWL Lite. In [de Bruijn, 2004] it is shown that this RDFS subset of OWL Lite is the same as the RDFS subset of OWL Lite-..

More precisely, the following features are RDFS are part of OWL Lite- too:

- Classes and class hierarchies
- Properties and property hierarchies
- Domain and range restrictions
- Individuals

The use of these features in OWL Lite is constraint in the same way as for OWL Lite [de Bruijn, 2004].

Comparing the expressivity of OWL-Lite- and RDFS, we can also identify an extension of the expressivity of RDFS. The features of OWL Lite- that are not in RDFS are, in short:

- Complete (i.e. necessary and sufficient) class definitions, whereas RDFS only allows partial class definitions.

---

<table>
<thead>
<tr>
<th>OWL Abstract Syntax</th>
<th>DL syntax</th>
<th>OWL Lite-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ⊆ C_i</td>
<td>+ (C_i ⊆ ⊥; A ⊆ ⊥)</td>
<td></td>
</tr>
<tr>
<td>A ⊆ C_1 ∩ ... ∩ C_n</td>
<td>+ (A, C_i ⊆ ⊥, ⊥)</td>
<td></td>
</tr>
<tr>
<td>A_1 ⊆ ... ⊆ A_n</td>
<td>++</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Language constructors of OWL Lite- and their relation to OWL Lite.
- Equivalence between classes and properties. In fact, this feature is semantically possible already in RDFS, but the syntax was missing.
- Inverse and symmetric properties.
- Transitive properties.
- Value restrictions in partial class definitions. In RDFS it is not possible to have local range restrictions for properties. In OWL Lite- it is, through the universal value restriction.

The features that are furthermore in OWL Lite, which are not in OWL Lite- are, besides the more general use of owl:Thing and owl:Nothing, (inverse) functional properties, (in)equality assertions of individuals, existential value restrictions, universal value restrictions in complete class definitions, minimal cardinality restrictions and maximal cardinality restrictions. But, as it is shown in [de Bruijn, 2004], most of these additional features are not intuitive in their usage and come at the cost of a significant increase in computational complexity.

Since OWL-Lite- is a proper subset of OWL Lite, there automatically is reasoning support for the new language: Every reasoner capable of dealing with OWL Lite descriptions can be used for reasoning over OWL Lite- specifications as well. Additionally, it is easily possible to construct new and very efficient reasoners (in particular for ABox reasoning) by exploiting a mapping which embeds OWL Lite specifications in Datalog. The mapping is show in Table 2.

Another interesting and important point in regard of the construction of OWL Lite- is the following: because OWL Lite- can be translated directly to Datalog, the language could be used as the basis for many extensions that have been investigated in the area of Logic Programming. Furthermore, after translating an OWL Lite- ontology to Datalog, building rules on top of the ontology is relatively straightforward.

Note also that, because it is not possible to derive negative information from a plain Datalog program, it is also not possible to derive negative information from an OWL Lite- ontology. In other words, it is not possible to have an inconsistency in an OWL Lite- ontology.

It’s important to notice, that although OWL Lite has some rudimentary support for concrete datatype, OWL Lite- has not. It clear, that support for concrete datatypes is important for many applications. The authors mention in [de Bruijn, 2004] that there will be extensions of OWL Lite-, for instance a language called OWL Flight, which supports concrete datatypes in a way that is more general than the support provided by OWL and much more useful for practical applications.

b. OWL-Flight

One of the important purposes with inventing OWL Lite- has been to come up with a clean conceptual starting point for a powerful and practically useful ontology language with efficient reasoning support.

OWL Lite- already overcomes some of the limitations of OWL Lite, but in some cases the expressivity had to be significantly reduced. For example, we had to leave out cardinality restrictions, because they introduce equality. Furthermore, many limitations of OWL Lite still exist in OWL Lite-, such as the lack of constraints and a sharp distinction between classes and instances. Also, OWL Lite- does not provide support for datatypes, which is commonly considered as essential for real-world applications.
For this reason, de Bruijn [de Bruijn, 2004] extends OWL Lite- with a number of features, such as support for datatypes, constraints and meta-classes to a language called OWL Flight.

In this section we will briefly overview this extension of OWL Lite-. The single added features in OWL Flight are:

- Datatype support
- Unique Name Assumption
- Constraints
- Classes-as-instances
- Local-closed world Assumption

**Datatype support.** The approach taken by OWL Flight for handling datatypes is based on the datatype group extension of OWL (called OWL-E) that has recently been proposed by J. Pan and I. Horrocks [Pan and Horrocks, 2004].

The datatype support provided by OWL Lite at present is only very limited: In order to support datatypes, an OWL Lite ontology is interpreted in two disjoint domains: the abstract domain and the concrete domain. An OWL Lite reasoner deals only with the abstract domain and assumes a datatype oracle, which is assumed to have a sound and complete decision procedure for the emptiness of an expression of the form $d_{D_1} \cup \ldots \cup d_{D_n}$ where $d_{D_i}$ is a (possibly negated) concrete data type from the domain $D$ [Horrocks and Sattler, 2001].

The three major limitations of datatype support in OWL are [Pan and Horrocks, 2004]:

- OWL does not support (general) negated datatypes.
- OWL does not support the use of datatype predicates. In OWL, it is only possible to refer to a single value in a datatype domain. It is, for example, not possible to express the greater-than relation for the `xsd:integer` domain in OWL. It is therefore not possible to express, for example, then an adult is at least 18 years old, although this is possible in most investigated Description Logic extensions with concrete domains (e.g. [Horrocks and Sattler, 2001]), where such an axiom can be easily expressed.
- OWL does not support user-defined datatypes. One would expect that OWL does support user-defined datatypes, especially because it uses the simple datatypes from XML Schema. In XML Schema it is possible for the user to define datatypes, however, these datatypes can not be used in OWL.

[Pan and Horrocks, 2004] shows an extension of OWL, called OWL-E, with so-called datatype groups. Datatype groups overcome the aforementioned limitations of datatype support in OWL and bridge the gap between datatypes in OWL and concrete domains as they have been investigated in the Description Logic community (see e.g. [Horrocks and Sattler, 2001]).

There currently exists a gap between the way data types are handled in OWL and the treatment of concrete domains in Description Logics. The latter only allows one concrete domain (e.g. `integer`), whereas the former allows many different data types (e.g. `string`, `date`, `integer`), albeit with many limitations. [Pan and Horrocks, 2004] shows a way to bridge the gap between the two, while extending datatype support in OWL, in order to allow the full expressiveness of concrete domains in Description Logics, while using different data types and retaining decidability. However, the datatype group approach still requires an external datatype oracle to evaluate the datatype expressions and, more specifically, to decide conjunctive queries for each datatype. Therefore, the authors of [de Bruijn, 2004] adopt the datatype groups extension proposed in [Pan and Horrocks, 2004] for OWL Flight.
**Unique Name Assumption.** The Unique Name Assumption can be easily introduced in a Description Logic knowledge base by including an inequality axiom $x \neq y$ for each pair of distinct individuals $x$ and $y$. Similarly, one can express this in OWL Lite using the `DifferentIndividuals(o1 ... oN)` statement, enumerating all distinct individuals $o1 ... oN$. When translating such a Description Logic knowledge base with the unique name assumption into a logic program, it is often not necessary to translate the inequality assertions, because logic programming engines and deductive databases typically adhere to the Unique Name Assumption. With the UNA syntactically different terms are assumed to be unequal. This means that unification only requires syntactic matching of terms, as opposed to checking satisfiability, which greatly speeds up the proof procedure.

Under the UNA, the semantics of OWL Flight diverge from the original semantics of OWL presented in [Patel-Schneider et al., 2003]. For the OWL Lite- subset of OWL Flight (and also OWL), this is no problem, since all features in OWL Lite, which relied on the absence of the UNA were purposely left out. In fact, for OWL Lite- it does not matter whether you implement the UNA or not.

OWL Flight adheres to the Unique Name Assumption in the sense that it is implicitly assumed that a statement `DifferentIndividuals(o1 ... oN)` where $o1 ... oN$ are all the individuals is in the knowledge base.

**Constraints.** In this section we present the different types of constraints of OWL Flight, all related to certain aspects of properties, such as cardinality and range. These constraints are related to the notion of integrity constraints. An integrity constraint is a rule without a head and is violated if all the literals in the body are true under a certain interpretation. Integrity constraints are directly supported in the Stable Model Semantics (SMS). When not using SMS, integrity constraints can be implemented by using a special predicate as the head of the rule of which the body is the integrity constraint. If the extension of the predicate is non-empty after the computation, the constraint is violated.

There are three types of constraints to be distinguished, namely minimal cardinality constraints, maximal cardinality constraints and value (range) constraints.

**Minimal cardinality constraints:** The specification of a minimal cardinality constraint of arity 1 requires Datalog(IC,not), that means function free Horn rules (Datalog) with Integrity Constraints (IC) and default negation (not). We illustrate the minimal cardinality constraint of property $R$ at class $C$, which is expressed as the following integrity constraint: 

$$\leftarrow \text{not } (R(?x,?y) \text{ and } C(?x))$$

A minimal cardinality constraint of arity higher than 1 additionally requires the use of inequality $\neq$ in the language and thus requires Datalog(IC,not, $\neq$). In order to check minimal cardinality constraints, some form of closed-world reasoning will be applied. This is different from the general OWL Semantics which is open-world.

**Maximal cardinality constraints:** In order to specify maximal cardinality constraints (of any arity), the language requires integrity constraints and inequality, thus Datalog(IC, $\neq$) suffices.

**Value constraints:** OWL (and also OWL Lite-) has universal value restrictions, which are actually assertions (that means we assume them to be true and derive things from this assumptions), rather than constraints. For example, restricting the range of property $R$ to class $D$ at class $C$ is written down (in first-order logic) in the following way: $D(?y) \leftarrow R(?x,?y) \text{ and } C(?x)$. Such an assertion would entail the fact that an instance is a member of $D$ if it is in the range of $R$. However, the authors of OWL Flight argue that it is more useful to check
whether an instance in the range of $R$ is actually a member of $D$. This can be done with the following integrity constraint: $\neg R(?x,?y)$ and $C(?x)$ and not $D(?y)$. Thus, we need both integrity constraints and default negation in our language, and thus Datalog(IC,not).

The use of default negation for these value constraints can bring us outside of the first-order style semantics of OWL. A form of non-monotonicity is introduced, because if we do not know that a certain value for a property is a member of $D$, the constraint is violated. However, if we later learn that this particular value is a member of $D$, the constraint is no longer violated, thus new information can invalidate existing inferences.

A more important issue is how to integrate value constraints into OWL Flight. Because the OWL Lite- semantics already guarantees that the value for the property is a member of $D$, simply adding the constraint to the knowledge base has no effect, since it will never be violated.

The solution to this problem is the introduction of a new type of value restrictions, alongside the existing value restrictions. The value restrictions coming from OWL Lite- are called *assertive* value restrictions. The value restrictions we introduced here are called *constraining* value restrictions. The designer of the ontology can choose the kind of value restrictions he/she wants to use. Clearly, when both an assertive and a constraining value restriction (assumed they both specify the same range) are specified for a certain property at a certain class, this amounts to an assertive restriction, since the constraint is then never violated. Because it does not make sense to model the same restriction both assertive and constraining, the ontology engineering tool should disallow this.

This approach guarantees a clean semantic layering on top of OWL Lite-, since the semantics of the value restrictions introduced in an OWL Lite- ontology do not change in OWL Flight. Thus, for an OWL Lite- ontology, the same conclusions will be drawn by both an OWL Lite- and an OWL Flight reasoner. However, if the OWL Lite- ontology is extended with this type of value constraints, existing inferences might be invalidated because of the non-monotonicity of this feature.

**Classes-as-instances.**

OWL Lite and OWL DL do not properly layer on top of RDFS. Several features of RDFS are not available in OWL Lite and DL. There are two properties of RDFS which complicate the layering of a Description Logic-style language on top of RDFS:

- Each RDF triple needs to have a meaning on its own
- RDF(S) meta-predicates, which are used for, for example, creating classes (rdfs:Class), subclassing (rdfs:SubClassOf) and typing instances (rdf:Type) are all part of the same domain as all the resources which are created by instantiating and subclassing these built-in resources.

With the latter it is possible, for instance to treat classes as instances. While it might seem awkward to, for example, subclass resources like rdf:Property or to add constraints to rdfs:Class, the meta-class facility of RDF(S) does seem useful on the Semantic Web [Schreiber, 2002].

Thus, OWL Flight will provide a meta-class facility, whereby the corresponding details are not fully worked out yet:
The authors identify three styles of semantics, which could be used for the meta-class facility in OWL Flight. They refer to these styles as the RDFS(FA) style, the HiLog style and the F-Logic style.

**RDFS(FA) style meta-classes**

RDFS(FA) [Pan and Horrocks, 2003] specifies a semantics for RDFS, which facilitates easy layering of Description Logic-based languages (e.g. OWL Lite/DL) on top of RDFS. The issue that the same resource in RDFS can be both a class and an instance and the issue that language constructs are part of the same domain of interpretation as the user-defined resources are resolved by defining strata for these resources.

RDFS(FA) divides the interpretation of an ontology into layers, or strata. The names in the strata are disjoint, i.e. a resource occurring in one stratum cannot occur in a different stratum. Furthermore, a resource occurring in a particular stratum always refers to a set of resources in the stratum directly below it. The stratum 0 contains all individual names. Stratum 1 contains classes in the Description Logic sense, i.e. sets of instances. Stratum 2 contains classes of classes, which corresponds to constructs such as rdfs:Class and rdf:Property.

Intuitively, in RDFS(FA), each terms corresponds to an instance of a class in the stratum directly above it and to a set (class) of instances in the stratum directly below it. Therefore, RDFS(FA) could possibly be used as the semantic basis for a classes-as-instances facility.

Because the strata are strictly separated, the same resource can never be an instance of itself and when reasoning with two adjacent strata at a time, standard Description Logic reasoning can be used.

A limitation is of course that one cannot treat a resource as both an instance and an object at the same time, but for many reasoning problems this is not a problem.

Extending OWL Lite- with RDFS(FA) requires some pre-processing when reasoning with a standard Datalog implementation. A Datalog implementation, which is extended with HiLog, such a FLORA-2 [Yang et al., 2003] could be readily used to handle such an extension.

**HiLog style meta-classes**

Because of its higher-order syntax, HiLog [Chen et al., 1993] can be easily used to extend a Description Logic-based language, such as OWL Lite- to include meta-classes. A limitation is that a formula in First-order logic (and thus Description Logic as well) and the same formula in represented in HiLog do not always have the same semantics. The semantics of FOL and HiLog (for a FOL formula) only coincide if the formula is cardinal, i.e. if the cardinality of the domain is at least as high as the number of symbols used in the language. This is the case for any set of equality-free sentences [Chen et al., 1993] and thus applying HiLog semantics to OWL Lite- does not change the semantics. Therefore, HiLog can be easily used to provide a meta-class facility for OWL Lite- and also for OWL Flight, as long as no equality is introduced in the language.

Using HiLog in the way the propose does not overcome the problem of having the RDFS constructs in the domain of interpretation. Therefore, in order to provide a clean layering on top of RDFS, a new semantics for RDFS is necessary, which differs both from the semantics proposed by Hayes [Hayes, 2004] and the semantics proposed by Pan and Horrocks [Pan and Horrocks, 2003].
F-Logic style meta-classes

When choosing an F-Logic [Kifer et al., 1995] style meta-classes facility, the semantics of the language need to be related to F-Logic. This can cause problems, because in F-Logic a class is not represented by a predicate, but by an object, which points to a set of objects. This allows for maximal flexibility in combining classes and instances. It allows a class to be defined as an instance of itself.

It is known that Description Logic languages can be axiomatized in F-Logic [Balaban; 1995]. However, this is based on the first-order style semantics for F-Logic originally specified in [Kifer et al., 1995]. Current implementations of F-Logic, such as FLORA-2 and OntoBroker, use a Logic Programming-style semantics. It is not known in how far Description Logics can be translated into this style of F-Logic. However, there does exist a full semantics-preserving translation from OWL Lite- to Datalog, which is subset of the current Logic Programming-style implementations of F-Logic and therefore, it is expected by the authors of OWL Lite-, that OWL Lite- can be axiomatized in F-Logic (LP), as well as OWL Flight, as long as it stay within the expressiveness of currently implemented logic programming formalisms.

Which approach will be taken in the future language definition is not clear at the moment, however it is clear, that there will be some meta-class facility in OWL Flight and that the possible approaches have been outlined.

Local-closed world Assumption.

The Closed-World Assumption (CWA) is typically applied in the semantics of logic programming (e.g. Prolog) and database applications. When applying the CWA you assume to have complete knowledge of the world, which implies that every fact that cannot be proven to be true is assumed to be false. This assumption can be very useful in order to infer new information from the absence of information. However, when knowledge is incomplete, this assumption might be inappropriate. This could be the case in the Semantic Web, because of its openness and distributiveness. A piece of information not known to an application might still be somewhere on the Semantic Web. This is why most current language for the Semantic Web, including OWL, adhere to the Open World Assumption (OWA). Under the OWA, it is only possible to infer negative information by explicitly stating it or by inferring it using some inference rules. In other words, under the OWA, a ground atomic sentence $S$ only holds when it is a direct consequence of the logical theory formed by the knowledge base $KB$, which consists of an ontology and a set on instances: $KB \text{ entails } S$.

Under the CWA, a ground atomic sentence $(\neg S)$ is assumed to be true when $S$ is not a logical consequence: $(KB \not\text{ entails } S) \text{ implies } (\neg S)$. In the Semantic Web setting the OWA might seem the way to go, because there can always be more information somewhere on the Web which you don't know about. However, in many cases it is necessary to do some form of closed-world reasoning, because otherwise some problems would become intractable or even unsolvable.

Notice also that the minimal cardinality and value constraints, introduced earlier in this chapter, require closed-world reasoning to check the constraints and thus implicitly apply the local closed-world assumption on the properties for which the constraints are specified.

Applying Local Closed-World (LCW) reasoning in the planning domain was investigated in [Etzioni et al., 1997]. [Heflin and Munoz-Avila, 2002] applies LCW-reasoning to two Semantic Web languages, namely SHOE and DAML+OIL. The latter was the basis for the current Semantic Web ontology language recommendation OWL.
[Etzioni et al, 1997] introduces the operator LCW, which allows to state that an agent has *local closed world information* (i.e. complete information) relative to a logical formula $F$. It represents some sort of integration between OWA and CWA.

For these reasons, the Local-Closed-World Assumption will be supported by OWL Flight.

The local closed-world assumption can be used in queries or definitions to specify complete knowledge for a certain predicate, which could be a class or a property.

We can now benefit from local closed-world reasoning by allowing negation of classes, which have been defined under the local closed-world assumption. This negation can now be treated as default negation, where it is up to the implementation to decide which type of negation to use (e.g. stratified negation, well-founded negation or negation under stable model semantics).

Note that the authors have already allowed default negation in cardinality constraints and value constraints, where they implicitly assume LCW for the properties in order to verify the constraints.

c. Further Extensions of OWL-Lite-: OWL-DL- and OWL-Full-

This section is not meant to actually describe further extensions of OWL Lite- in detail but just serves for making aware, what future activities are to be expected in the course of the WSMO project with respect to ontology languages. The reason for this is that at present, there are no publicly available deliverables which officially report about the languages mentioned here and we only know about these activities because of private communication with Jos de Bruijn and attending WSMO and WSML phone conferences.

At present it is planned (but not really precisely defined how) to extend OWL Lite- in two more languages: OWL-DL- and OWL-Full-.

**OWL-DL-**

In section (a) we have overviewed OWL Lite- as the subset of OWL Lite, which can be evaluated on a Datalog engine. OWL-DL- now will extend OWL Lite- to OWL DL- following the general idea to add as many features from OWL-DL in OWL-Lite- such that the language still can be mapped to Datalog in order to allow efficient reasoning and query answering even with large sets of instances.

The *SHIF* Description Logic underlying OWL Lite does not allow the use of individuals in concept descriptions. Therefore, the *hasValue* property restriction, which is in OWL DL, is not in OWL Lite. However, as was shown by [Volz, 2004], the *hasValue* restriction does not introduce additional complexity when performing query answering in deductive databases. Therefore, it might be useful to re-introduce this restriction. One could also think of using the *hasValue* restriction as the basis for the use of (overridable) default values of properties. The *hasValue* restriction is allowed in both, partial and complete class definitions.

A second line of extensions could be concerned with making hidden features of explicit in OWL Lite: Many features of the OWL DL species that are not explicitly present in OWL Lite can be expressed in OWL Lite via simple transformations. In fact, the only descriptions in OWL DL that cannot be expressed in OWL Lite are those containing individuals and cardinalities higher than one [Horrocks et al., 2003].
Some of these features, but by no means all, can also be expressed in OWL Lite. For example, intersection can be expressed by creating a class definition from more than one class or property restriction. As shown in [de Brujin, 2004], this translates to an intersection statement in Description Logic syntax. A possible (syntactic) extension of OWL Lite is to enrich the syntax to make these hidden features explicit in the language. A possible drawback of this could be that the language might become too complex for its users, which was also the original argument for leaving these features out of the syntax of OWL Lite.

**OWL-Full**

OWL Full will represent an extension of OWL DL with a meta-class facility and thus bridging the gap between OWL-DL and RDFS. Given the available documents at present, it is not completely clear, how the actual relationship between OWL-Full and OWL-Flight will look like. We suspect that the main distinguishing features could be:

- Unique Name Assumption
- Support for Constraints
- Local-closed world Assumption

We hope to be able to report more details as soon as the according deliverables are available. Thus, some more material will be included at the latest in the next version of this deliverable.

d. **WSML-Core**

In addition, to the OWL family of ontology languages presented in Sections (a) and (c), there is an ontology language called WSML-Core currently being developed in the WSML working group which essentially combines two things: OWL Lite and the conceptual meta-model for ontologies as defined in the WSMO Ontology [Roman et al., 2004]. Roughly speaking, we can say that OWL-Lite provides the semantic basis for WSML-Core whereas the conceptual model for ontologies of WSMO provides the basic modelling elements of the language.

WSML-Core represents the most basic language that can be used for describing the semantics of web services. It represents the intersection between two prominent knowledge representation paradigms, namely Description Logics and Rule Languages. Thus, this language serves a starting point for extension both directions.

**I) Basic WSML-Core Syntax**

The syntax for WSML-Core has a frame-like style, where a collection of information about a class or property is given in one large syntactic construct, instead of being divided into a number of atomic chunks or even being divided into several triples (as when writing using an RDF syntax). It is possible to spread the information about a particular class, relation, instance or axiom over several constructs, but it is not recommended to do so.

Actually, WSML-Core is inspired by F-Logic and has strong syntactic similarities. The same will hold for the further upcoming languages in the WSML project.

Identifiers in WSML-Core follow the conventions of WSMO [Roman et al., 2004]. An identifier is either a URI or a Qualified Name (QName). QNames without a prefix are resolved with the default namespace. Argument lists are separated by commas. Statements in WSML-Core start with a keyword and can be multi-lined. There is no specific end-of-statement syntax -- a keyword for a new statement is sufficient. In the following paragraphs
we will briefly describe how to specify ontologies in WSML based on the conceptual model for ontologies in WSMO.

**Ontologies**

An ontology definition in WSML-Core starts with the **ontology** keyword followed by a full URI, which serves as the identifier of the ontology and, when there is no explicit target namespace definition, as the target namespace of the definitions in the ontology specification document. An example:

```
ontology http://www.wsmo.org/2004/d3/d3.2/v0.1/20040628/resources/po.wsml
```

**Namespace definitions**

Below the **ontology** URI declaration, there is an optional block of namespace definitions, which is preceded by the **namespace** keyword. The **namespace** keyword is optionally followed by the default namespace and a number of namespace definitions. Each namespace definition, except for the default namespace, consists of the chosen prefix, an '=' and the URI, which identifies the namespace. Finally, the (optional) target namespace is specified with the use of the **target-namespace** keyword. If the target namespace is designated by the URI which is specified by the **ontology** declaration, the **target-namespace** line is redundant and may be omitted. An example:

```
namespace
    http://www.wsmo.org/ontologies/purchase#,
    dc=http://purl.org/dc/elements/1.1#,
    wsml=http://www.wsmo.org/2004/d2/v1.0#
    target-namespace http://www.wsmo.org/ontologies/purchase#
```

**Non-functional properties**

Following the **namespace** definition block is an optional non-functional properties definition block, identified by the keyword **non-functional-properties**. Following the keyword is a list of properties and property values. The recommended properties are the properties of the Dublin Core [Weibel et al. 1998], but the list of properties is extensible and thus the user can choose to use properties coming from different sources. WSMO defines one property, absent in the Dublin Core, namely **version**. The value of each of the properties is either a URI or a literal. If a property has multiple values, these are separated by commas. An example:

```
non-functional-properties
    dc:title "Purchase ontology"
    dc:creator "DERI International"
                "Payment method", "Delivery"
    dc:description "General purchase order request ontology"
    dc:publisher "DERI International"
    dc:contributor "Armin Haller"
```
Import Ontology definitions

Following the **non-functional-properties** definition block is an optional imported ontologies definition block, identified by the keyword **import-ontologies**. Following the keyword is a list of URIs identifying the ontologies being imported. An **import-ontologies** definition serves to merge ontologies, just as an OWL **import** statement does. This means the logical definitions in the imported ontologies are simply appended to the definitions of the importing ontology. An example:

```
import-ontologies

http://www.wsmo.org/ontologies/dateTime#,
http://www.wsmo.org/ontologies/currency#
```

Mediators

Mediators are used to import other ontologies. This concept of mediation between ontologies is more flexible than the **import-ontologies** statement, which is used to import an WSML ontology into another WSML ontology. The ontology import mechanism simply appends the definitions in the imported ontology to the importing ontology. The importing ontology should contain the axioms to relate the definitions in the imported ontology to the local definitions. This mechanism enforces a strong coupling between the ontologies, which is undesirable in the general case and it does not allow importing parts of ontologies. By externalizing the mediation from the ontology, WSMO allows loose coupling of ontologies; the mediator is responsible for relating the different ontologies to each other. A mediator can provide, for example, translation services between ontology languages or adjust for argument-order differences. Mediators are identified and referred to by using URIs. An example that declares the use of two specific mediators (described in the corresponding WSML files):

```
usedMediators

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040628/resources/owlFactbookMediator.wsml,
http://www.wsmo.org/2004/d3/d3.2/v0.1/20040628/resources/owlAddressMediator.wsml
```

The reuse of mediators is possible in principle, but it heavily depends on the kind of mediations that is necessary: Mediators can be reused for the same ontologies (they refer to)
whenever these ontologies are used (which is a rather specific form of reuse). If the kind of mediation that is performed is rather generic and does not apply to specific ontologies only, then the mediator can even be reused for different ontologies (of the same sort). The latter case could apply for instance to mediators which convert ontologies represented in different ontology languages in some common format. Nonetheless, the reuse of mediators is not always possible in this general way, since they might capture very specific knowledge about how to integrate specific domain-depended ontologies which is not applicable to other domains. Still, in this case they can be reused within the addressed problem domain.

**Concept definitions**

A concept definition starts with the `concept` keyword, which is followed by the identifier of the concept. This is followed by an optional `non-functional-properties` block, zero or more (direct) superconcept definitions (using the `subconcept-of` keyword followed by the identifier for a named concept), and zero or more attribute specifications. An attribute specification consists of the identifier of a relation, the `oftype` keyword and the identifier of a concept of which the attribute values must be instances.

A concept can only be a `subconcept-of` named concepts. We do not allow complex concepts definitions as superconcepts, as is allowed in OWL. An example:

```plaintext
concept Human
  non-functional-properties
    dcterms:description "Members of the species Homo sapiens"
    subconcept-of Primate
    subconcept-of LegalAgent
  parentOf oftype Human
```

**Relation definitions**

A relation definition starts with the `relation` keyword, which is followed by the identifier of the relation and an optional specialization of a direct super-relation. This is followed by an optional `non-functional-properties` block. In WSML-Core, relations are restricted to binary relations, which correspond with object properties in OWL Lite. Parameters can be used to restrict the domain and range of the property (denoted by the `domain` and `range` keywords, respectively). It fact, no other parameters are allowed.

Particular aspects of the relation are denoted by particular keywords. The relation can be a specialization of a different named relation, denoted with the `subrelation-of` keyword. Transitive, symmetric, and inverse properties are denoted with the `transitive`, `symmetric`, and `inverse-of` keywords, respectively. An example:

```plaintext
relation hasAncestor subrelation-of hasRelative
  non-functional-properties
    dcterms:description "Relation between ancestors"
    transitive
  domain oftype Person
  range oftype Person
```
Instance definitions

An instance definition starts with the instance keyword, followed by the identifier of the instance, the member-of keyword and the name of the concept to which the instance belongs. An instance corresponds with an individual in OWL Lite. The member-of keyword identifies the concept to which the instance belongs. This definition is followed by the attribute values associated with the instance. Each property filler consists of the property identifier, the keyword hasvalues and the value for the attribute. An example:

```java
instance innsbruckHbf member-of station
  name hasvalues IBKHbf
  code hasvalues INN
  locatedIn hasvalues loc:innsbruck
```

The second way of accessing instances, as outlined in [Roman et al., 2004], is by providing a link to an instance store. Instance stores contain large numbers of instances and they are linked to the ontology with the use of a mediator.

Axiom Definitions

An axiom definition starts with the axiom keyword, followed by the name (identifier) of the axiom. This is followed by an optional non-functional-properties block) and then by a logical expression which is preceded by the logical-expression keyword. An example:

```java
axiom example-axiom
  logical-expression
    lessThan(?xDistance, ?yDistance) <-
    meters(?xDisance, ?xMeters) and
    meters(?yDisance, ?yMeters) and
    lessThan(?xMeters, ?yMeters)
```
(II) WSML-Core Semantics: Mapping WSML-Core to OWL Lite  

In this paragraph we overview the semantics of WSML-Core by giving a mapping to the OWL Lite abstract syntax [de Bruijn, 2004], which is a subset of the OWL abstract syntax. The syntax and semantics of the OWL abstract syntax can be found in [Patel-Schneider, 2003].

Table 1 shows the mapping between the WSML-Core conceptual syntax and OWL Lite. Through the mapping to the OWL abstract syntax, the precise semantics of the WSML-Core primitives is defined. Please note in WSML-Core, each construct can have non-functional properties associated with it. This is not reflected in the table. Rather, there is a separate row in the table, which addresses the non-functional properties and the way they are reflected in the OWL abstract syntax. The annotation properties occur inside each class, property or instance definition.

<table>
<thead>
<tr>
<th>WSML-Core conceptual syntax</th>
<th>OWL Lite Abstract syntax</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logical Definitions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>**ontology</td>
<td>**</td>
<td>Ontology(O)</td>
</tr>
<tr>
<td><strong>concept C</strong></td>
<td></td>
<td>Class(C partial $D_1, \ldots, D_n$) restriction($P_1$ allValuesFrom $C_1$) \ldots restriction($P_n$ allValuesFrom $C_n$))</td>
</tr>
<tr>
<td><strong>relation R</strong> [subrelation-of $R_1, \ldots, R_n$] [transitive] [symmetric] [inverse-of($R_0$)] [domain oftype $C_1, \ldots, C_n$] [range oftype $D_1, \ldots, D_n$]</td>
<td>ObjectProperty($R$ super($R_1$) \ldots super($R_n$)) [Transitive] [Symmetric] [inverseOf($R_0$)] domain($C_1$) \ldots domain($C_n$) range($D_1$) \ldots range($D_n$)</td>
<td></td>
</tr>
</tbody>
</table>
### Extra-Logical definitions

**non-functional-properties**

\[
\begin{align*}
P_1 v_1 \\
. \\
. \\
P_n v_n
\end{align*}
\]

\[
\text{version } v_0
\]

annotation\((P_1 v_1)\),

\[
. \\
. \\
\text{annotation}\((P_n v_n)\)
\]

[annotation(owl:versionInfo \(v_0\))]

**import-ontologies**

\[
O_1, \\
. \\
. \\
O_n
\]

**usedMediators**

\[
M_1, \\
. \\
. \\
M_n
\]

---

**Table 1: Mapping between WSML-Core and OWL Lite abstract syntax**

1. **Instance** \(o\) member-of \(C_1, \ldots, C_n\)
2. **A1 hasvalues** \(o_1\)
3. **An hasvalues** \(o_n\)
4. **Individual** \((o \text{ type}(C_1) \ldots \text{type}(C_n))\)
5. **value** \((A_1 o_1 \ldots \text{value}(A_n o_n))\)

**Annotation properties** are nested inside other definitions, such as ontology, individual, class and property definitions. For some strange reason, if annotations occur on the ontology level, they should be written with a capital 'A' (e.g. Annotation(owl:versionInfo "$Revision: 1.50$")).

**import-ontologies**

The OWL abstract syntax does not provide a construct for the import of ontologies, although the RDF/XML serialization does provide this facility through the import statement.

**usedMediators**

OWL does not have the concept of a mediator. Therefore, this construct cannot be translated to OWL.
We need to make two notes here about the WSML-Core syntax compared to the syntax presented in WSMO-Standard [Roman et al., 2004]. First of all, the attribute definitions in the concept signature (see the second row of Table 1) are interpreted as universal value restrictions in OWL Lite. This does not correspond with the intuition behind these attribute definitions, as it was pointed out in [de Bruijn, 2004]. Second, on the one hand the WSMO-Standard syntax for relations is restricted by allowing only two parameters (domain and range) in order to restrict the binary relations. On the other hand, the WSMO-Standard syntax is extended with the keywords subrelation-of, transitive, symmetric, and inverse-of in order to capture the modelling elements of the ObjectProperties in OWL.

In Table 2, we have shown exactly which logical expressions can be translated to OWL Lite. These logical expressions add little in expressivity over the conceptual syntax, but sometimes the user prefers a different way of modeling axioms. Furthermore, this syntax for logical expressions can be directly used in goal descriptions and capability descriptions of web services for the modeling of assumptions, pre-conditions, effects and post-conditions. All and only those logical expressions that are either in the first column of Table 2 or can be reduced to logical expressions in Table 2 are valid in WSML-Core.

<table>
<thead>
<tr>
<th>WSML-Core Logical Expression</th>
<th>OWL Lite Abstract Syntax</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C subconcept-of D</td>
<td>Class (C partial D)</td>
<td></td>
</tr>
<tr>
<td>?x member-of D &lt;-&gt; ?x member-of C1 and ... and ?x member-of Cn</td>
<td>Class (D complete C1 ... Cn)</td>
<td></td>
</tr>
<tr>
<td>C[R oftype D]</td>
<td>Class (C partial restriction(R allValuesFrom D))</td>
<td></td>
</tr>
<tr>
<td>R subrelation-of S</td>
<td>ObjectProperty (R super(S))</td>
<td></td>
</tr>
<tr>
<td>?x member-of C &lt;-&gt;</td>
<td>ObjectProperty (R domain(C))</td>
<td></td>
</tr>
<tr>
<td>Logical Expression</td>
<td>OWL Lite Syntax</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>?y member-of C</code></td>
<td><code>ObjectProperty (R range(S))</code></td>
<td></td>
</tr>
<tr>
<td><code>R(?x,?y)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>S(?y,?x)</code></td>
<td><code>ObjectProperty (R inverseOf(S))</code></td>
<td></td>
</tr>
<tr>
<td><code>R(?x,?y)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>R(?y,?x)</code></td>
<td><code>ObjectProperty (R Symmetric)</code></td>
<td></td>
</tr>
<tr>
<td><code>R(?y,?x)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>R(?x,?y)</code></td>
<td><code>ObjectProperty (R Transitive)</code></td>
<td></td>
</tr>
<tr>
<td><code>R(?x,?y)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>o member-of C</code></td>
<td><code>Individual(o type(C) value(R1 o1) ... value(Rn on))</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Both the <code>member-of C</code> and all the property values are optional.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Mapping between WSML-Core logical expressions and OWL Lite' abstract syntax

Obviously, the most fundamental limitation as an ontology language that WSML-Core inherits from OWL-Lite- is the lack of support for concrete datatypes, like integer and strings.
Summary

In the course of the WSMO initiative there are some recent activities in creating a series of useful and practical ontology languages which try to overcome the deficiencies of the OWL family of ontology languages.

At present, there is only a precise definition and discussion for OWL-Lite-, a proper subset of OWL Lite with nice computational and extendability properties: There exists a direct translation into the deductive database language Datalog. Thus, any OWL Lite- ontology can be translated into Datalog in order to allow for efficient query answering. It turns out that most current ontologies fall inside this fragment.

An ontology language for which a translation to Datalog exists has several advantages. Most notably, it can benefit from highly optimized query answering engines and allows for easy implementation of a rule and a query language on top of the ontology.

Since OWL Lite- is a subset of OWL Lite, all tools that can deal with OWL Lite- can automatically be used for OWL Lite-. Moreover, by implementing the translation from OWL Lite- to Datalog, one can use any existing Datalog engine for efficient reasoning on ontologies and large sets of instances.

OWL Flight represents an extension of OWL Lite- with datatype support, Unique Name Assumption, Constraints, Classes-as-instances (Metaclasses, resp.), and Local-closed world Assumption. Currently, the language is not fully defined in all aspects and there are no reasoning tools available.

OWL DL- and OWL Full- are intended extensions of OWL with attractive computational properties. These languages can only be sketched here since there are no official documentation and discussion available at the moment.

WSML-Core is a separate language that is based on OWL-Lite- and combines OWL-Lite- with the ontology model described in the WSMO ontology. In principle, this adds additional expressivity to OWL-Lite-. Currently, there is no implementation of reasoning support for WSML-Core, since the language is still in the process of being developed.

The WSMO initiative is a rapidly evolving initiative where new deliverables within the project popup regularly in intervals of weeks. From private communications with the authors of OWL Lite- as well as attendance of the weekly WSMO and WSML phone conferences, we can foresee, that there will be a bunch of new interrelated ontology languages as well as reasoning support during the next months.

One of the final aims of the initiative is to eventually start standardization processes for these languages like the one followed with the OWL family of languages in the context of W3C. Clearly, standardization is necessary for the final goal of the WSMO initiative to provide a powerful, flexible and extendable framework and tools for the description and use of Semantic Web Services that serves as the basis for a standard in this field.

Resources

http://www.wsmo.org
http://www.wsmo.org/wsml
http://www.wsmo.org/2004/d20/d20.1/v0.2/20040719/
http://www.w3.org/TR/rdf-owl/
2.5.1 Updates and improvements: August 2004-January 2005

The WSMO initiative as well as the related initiatives WSML and WSMX are very active working groups that develop and evolve conceptual models for Semantic Web Services, description languages for them as well as concrete technology which allows to exploit these descriptions in an execution engine for Semantic Web Services.

For this reason, some of the languages that we discussed already earlier evolved and changed a bit and a few new languages have been designed by the WSML working group. We will discuss the new languages as well as the changes here only briefly. A more detailed account can be found in the WSML Deliverable D16.1 which is included in the Resources Section.

At present it is as well clear that there are some more languages are to be expected by the WSML working group, namely WSML DL, WSML Rule and WSML Full. The purpose of these languages is – besides being languages for more expressive ontologies – to provide a thorough basis for an ontology-based description of Semantic Web Services and their behaviour. Unfortunately, these languages are not defined by now, only a few ideas exist what features these languages could eventually provide. Thus, these languages can not be discussed here, although they might be very interesting for the Esperonto project and projects in the Semantic Web arena as well.

2.5.1.1 Overview on the WSML family of languages

The Web Service Modeling Ontology WSMO proposes a conceptual model for the description of Ontologies and Semantic Web Services and Ontologies, providing the conceptual grounding for Ontology and Web Service description. The WSML working group currently develops a family of representation languages which take the conceptual model of WSMO as a starting point for the specification of a Web Service description and Ontology specification languages. The single languages have the prefix “WSML-” in common, which is followed by an identifier for a specific variant of WSML.

The different variants of WSML correspond with different levels of logical expressiveness and the use of different styles of logical languages. More specifically, the working group takes Description Logics, First-Order Logic and Logic Programming as starting points for the development of the various WSML variants. We provide a clean layering of the WSML variants, both syntactically and semantically. All WSML variants are specified in terms of a human-readable syntax with keywords similar to the elements of the WSMO conceptual model. Furthermore, an XML syntax as an exchange language, as well as an RDF syntax and a mapping to OWL for interoperability with RDF- and OWL-based applications is provided.

For the sake of space, we will not give all details here, but instead refer to the still evolving deliverable document D16.1 “The WSML Family of Representation Languages” which can be found at http://www.wsmo.org/2004/d16/d16.1/ (always in the latest version!). Here we refer to the deliverable version http://www.wsmo.org/2004/d16/d16.1/v0.2/20041224/.

In Figure 1 the different variants of WSML and the relation between them are shown. In the figure, an arrow stands for "extension in the direction of". The variants differ in the logical expressivity they offer, and thus in the computational complexity for different reasoning tasks. By offering these variants, we allow users to make the trade-off between the provided expressivity and the implied complexity on a per-application basis. As can be seen from the figure, the basic language WSML-Core is extended in
two directions, namely Description Logics (WSML-DL) and Logic Programming (WSML-Flight, WSML-Rule). WSML-Rule and WSML-DL are both extended to a full First-Order Logic with nonmonotonic extensions (WSML-Full), which unifies both paradigms.

WSML-Core
This language is defined by the intersection of Description Logic and Horn Logic, based on [Grosos et al., 2003]. It has the least expressive power of all the languages of the WSML family and therefore the most preferable computational characteristics. The main features of the language are the support for modeling classes, attributes, binary relations and instances. Furthermore, the language supports class hierarchies, as well as relation hierarchies. WSML-Core provides extensive support for datatypes and datatype predicates, as well as user-defined datatypes [2]. WSML-Core is based on a subset of OWL, called OWL- [de Bruijn et al., 2004], with a datatype extension based on OWL-E [Pan and Horrocks, 2004], which adds richer datatype support to OWL.

WSML-Flight
This language is an extension of WSML-Core with several features from OWL Full, such as meta-classes, and other features, such as constraints and nonmonotonic negation. WSML-Flight is based on OWL Flight [de Bruijn et al., 2004a], which adds features such as constraints and meta-classes to a subset of OWL DL. Furthermore, WSML-Flight is based on a logic programming variant of F-Logic [Kifer et al., 1995].

WSML-Rule
This language will be an extension of WSML-Flight in the direction of Logic Programming. The language will capture several extensions such as the use of function symbols and possibly extensions based on HiLog [Chen et al., 1993] and Transaction Logic [Bonner & Kifer, 1998], which are required for rich Web Service discovery [Keller et al., 2004]. Another possible extension for WSML-Rule is disjunction in the head of the rule, as in Disjunctive Datalog [Eiter et al., 1997].
WSML-DL

This language is an extension of WSML-Core which fully captures the Description Logic $SHOIN(D)$, which underlies the (DL species of the) Web Ontology Language OWL [Dean & Schreiber, 2004]. The language can be seen as an alternate syntax for OWL DL, based on the WSMO conceptual model.

WSML-Full

WSML-Full unifies WSML-DL and WSML-Rule under a First-Order umbrella with extensions to support specific nonmonotonic features of WSML-Rule, such as minimal model semantics and default negation. It is yet to be investigated which kind of formalisms are required to achieve this.

Figure 2. WSML Layering

Figure 2 illustrates the layering of WSML languages. As can be seen from the figure, there are two alternative layerings, namely WSML-Core -> WSML-Flight -> WSML-Rule -> WSML-Full and WSML-Core -> WSML-DL -> WSML-Full. In both layerings, WSML-Core is the least expressive and WSML-Full is the most expressive language. The two layerings are to a certain extent disjoint in the sense that interoperation between the Description Logic variant (WSML-DL) on the one hand and the Logic Programming variants (WSML-Flight and WSML-Rule) on the other, is only possible through a common core (WSML-Core) or through a very expressive (undecidable) superset (WSML-Full). However, there are proposals which allow interoperation between the two while retaining decidability of the satisfiability problem, either by reducing the expressiveness of either of the two paradigms, thereby effectively adding expressiveness of either of the two paradigms to the intersection (cf. [Levy & Rousset, 1998]) or by reducing the interface between the two paradigms and independently reason with both paradigms (cf. [Eiter et al., 2004]).[1]

The only languages currently specified by the WSML working group are WSML-Core and WSML-Flight. WSML-Rule is currently underspecified, because the requirements
on the language are not yet clear. WSML-DL will correspond (semantically) with the Description Logic \( \text{SHOIN}(\mathcal{D}_n) \), extended with more extensive datatype support. WSML-Full will provide the formal semantics for the logical language specified in WSMO D2 [Roman et al., 2004].

In the remaining parts of this Section we will give a description of the general syntax which is shared by the single WSML variants. Additionally, we describe two specific variants of WSML, namely WSML-Core and WSML-Flight, as they are defined at present. In particular, we described the specific restrictions and extensions that these language variants use with respect to the common shared WSML syntax.

2.5.1.2 Common base syntax

Before we introduce the individual WSML variants, we introduce the elements they have in common. In this section we introduce the basic structure of WSML specifications, and the use of various WSML elements which the different variants have in common.

This Section is structured as follows. We first introduce basics of the WSML syntax, such as the use of namespaces, identifiers, etc. Then we describe the elements all WSML variants have in common.

I – Basics of the WSML syntax

The syntax for WSML has a frame-like style. The information about a class and its attributes, a relation and its parameters and an instance and its attribute values is specified in one large syntactic construct, instead of being divided into a number of atomic chunks. It is possible to spread the information about a particular class, relation, instance or axiom over several constructs, but we do not recommend this. In fact, in this respect, WSML is similar to OIL [Fensel et al., 2001], which also offers the possibility of either grouping descriptions together in frames or spreading the descriptions throughout the document. One important difference with OIL (and OWL) is that attributes are defined locally to a class and should in principle not be used outside of the context of that class and its subclasses.

Nonetheless, attribute names are global and it is possible to specify global behavior of attributes through logical expressions. However, we do not expect this to be necessary in the general case and we strongly advise against it.

Argument lists in WSML are separated by commas and surrounded by curly brackets. Statements in WSML-Core start with a keyword and can be spread over multiple lines.

A WSML specification is separated in two parts. The first part provides meta-information about the specification, which consists of such things as WSML variant identification, namespace references, entity header (non-functional properties (annotations), import of ontologies and references to used mediators) and the type of the specification. This meta-information block is strictly ordered. The body of the specification, consisting of elements such as concepts, attributes, relations (in the case of an ontology specification), capability, interfaces (in the case of a web service specification), etc., is not ordered.
The remainder of this section explains the use of namespaces in WSML, identifiers in WSML and datatypes in WSML. Subsequent sections of this chapter will explain the different kinds of WSML specifications and the basic WSML logical expression syntax.

(a) Namespaces in WSML

WSML inherits the namespace mechanism of WSMO, which is inherited from XML. The WSML keywords themselves have the namespace http://www.wsmo.org/2004/wsml.

Namespaces can be used to syntactically distinguish elements of multiple WSML specifications, and more general, resources on the Web. A namespace is a syntactical domain. Each element specified in a WSML document inherits this namespace from the overall document and the complete identifier of the element corresponds with the concatenation of the namespace of the document with the local name of the element. Note that namespaces only provide a syntactical separation of names.

Each element in a WSML ontology is created in the default namespace of the ontology, which is an IRI (see also the next Section).

Whenever an ontology has a specific identifier, it is good practice to have a relevant document on the location to which the identifier refers. This can either be the WSML document itself or a natural language document related to the WSML document. Note that the identifier of an ontology does not have to coincide with the location of the ontology. It is good practice, however, to include a related document, possibly pointing to the WSML specification itself, at the location pointed to be the identifier.

(b) Identifiers in WSML

An identifier in WSML is either an IRI [Duerst & Suignard, 2004], literal or an anonymous id. In logical expressions variables can as well be used as identifiers (see e.g. also section 3.6).

Internationalized Resource Identifiers

The IRI (Internationalized Resource Identifier) mechanism provides a way to identify resources. IRIs may point to resources on the Web (in which case the IRI can start with 'http://'), but this is not necessary (e.g. books can be identified through IRI starting with 'urn:isbn:'). The IRI proposal is a successor to the popular URI standard. In fact, every URI is an IRI. An IRI can be abbreviated to a QName. A QName consists of two parts, namely the namespace prefix and the local part, separated with a colon (':'). A QName is equivalent to the IRI which is obtained by concatenating the namespace (to which the prefix refers) with the local part of the QName. Therefore, a QName can be seen as an abbreviation for a IRI which enhances the legibility of the specification. In case a QName has no prefix, the namespace for the QName is the default namespace of the document. We adopt the syntax for QNames from XML.

A full IRI in WSML is enclosed in double angle brackets ‘<'' and ‘''>. For convenience, a QName does not require special delimiters. However, QNames may not
coincide with any WSML keywords. The characters '.' and '-' in a QName need to be escaped using a '\':

```plaintext
full_iri = '<' iri_reference '>'
qname = (ncname ':')? ncname
```

Please note that the IRI of a resource does not necessarily correspond to its location on the Web. Therefore, we distinguish between the identifier and the locator of a resource. The locator of a resource is an IRI which can be mapped to a location from which the (information about the) resource can be retrieved.

**Literals**

*Literals* in WSML are identifiers of concrete data values. Literals may be typed or untyped. A type in this case represents a particular value domain (e.g. integer).

A literal is a Unicode sequence in normal form C, enclosed in double quotes "". In case a literal is typed, this type is indicated with the double caret ^^, e.g. "4"^^xsd:integer stands for the integer 4. Double quotes inside a literal should be escaped using the escape character "\" ("backslash"): ". The backslash "\" can be escaped itself: \". As types for literals we recommend to use the XML Schema Datatypes [Biron & Malhorta, 2004], usually referred to with the xsd namespace prefix. For more information about types of literals (datatypes), see the next section. Finally, a plain literal may have a language tag associated with it, according to RFC3066: http://www.ietf.org/rfc/rfc3066.txt

WSML allows the following syntactical shortcuts for literals:

- Literals of type xsd:string can be written between single quotes '. Thus a literal of the form 'string' is a shortcut for "string"^^xsd:string. Single quotes inside a string should be escaped using the "\" ("backslash"): ".
- Literals of type xsd:integer can be written by simply omitting the single quotes and the datatype. Thus a literal of the form integer is a shortcut for "integer"^^xsd:integer. For example, 4 is a shortcut for "4"^^xsd:integer
- Literals of type xsd:float can be written by simply omitting the single quotes and the datatype. Thus a literal of the form float is a shortcut for "float"^^xsd:float. For example, 4.2 is a shortcut for "4.2"^^xsd:float

```plaintext
literal = {typedliteral} typedliteral
          | {plainliteral} plainliteral
          | {numeric} number
          | {string} string

plainliteral = '"' literal_content* '"' ( language_tag )?

typedliteral = plainliteral dblcaret iri
```
**Variables**

Variable names start with an initial question mark, "?". Variables may occur in place of concepts, attributes, instances, attribute values, or literals. A variable may however not replace a WSML keyword. Furthermore, variables may only be used inside logical expressions.

The scope of a variable it always defined by its quantification. If a variable is not quantified inside a formula, the variable is free.

```
variable = '?' alphanum+
```

**Anonymous identifiers**

Anonymous identifiers in WSML follow the naming convention for anonymous IDs presented in [Yang & Kifer, 2001]. Unnumbered anonymous IDs are denoted with ‘_#’. Each occurrence of ‘_#’ denotes a new anonymous ID and different occurrences of ‘_#’ are unrelated.

Numbered anonymous IDs are denoted with ‘_#n’ where n stands for an integer denoting the number of the anonymous ID. All occurrences of a particular numbered anonymous ID in the same scope refer to the same object. Each occurrence of an unnumbered anonymous ID can be seen as a new unique identifier. Each numbered anonymous ID can be seen as a new unique identifier which shared inside its scope.

In order to determine the scope of a particular numbered anonymous ID we need to define the notion of a scope in WSML. The largest scope of an anonymous ID is a single WSML document. This scope is shared between the header elements of the specification. Nested inside this scope are the elements of a particular ontology, goal, mediator or web service. Finally, each of these elements has a local scope with respect to the attributes, parameters, etc. The smallest scope is the logical expression. Each logical formula has a local scope.

Certain occurrences of unnumbered anonymous IDs (’_#’) can be disregarded, namely when the unnumbered anonymous ID is an identifier of a relationInstance, since an instance of a relation simply consists of a tuple, identified by its parameter values.

```
anonymous = ' #' digit*
```

The use of an identifier in the specification of WSML elements is optional. In case no identifier is specified, the following default rules apply:

- In case the identifier of an ontology, web service, goal or mediator is omitted, the identifier is assumed to be the same as the locator of the specification, i.e. the location where the specification can be found. Notice that in case no explicit target namespace has been specified, that this is also the target namespace of the specification.
- In case the identifier of a WSML element (e.g. concept, relation, postcondition) has been omitted, the anonymous identifier ‘_#’ is used to identify the element.
In case the same identifier is used for different definitions, it is interpreted differently, depending on the context. In a concept definition, an identifier is interpreted as a concept; in a relation definition this same identifier is interpreted as a relation. If, however, the same identifier is used in separate definitions, but with the same context, then the interpretation of the identifier has to conform to both definitions and thus the definitions are interpreted conjunctively. For example, if there are two concept definitions which are concerned with the same concept identifier, the resulting concept definition includes all attributes of the original definitions and if the same attribute is defined in both definitions, the range of the resulting attribute will be equivalent to the conjunction of the original attributes.

**Definition 1.** A WSML vocabulary $V$ consists of:

- A set of identifiers $V_{ID}$
- A set of abstract identifiers $V_{AID}$ which is a subset of $V_{ID}$. $V_{AID}$ consists of all IRI references, QNames and anonymous identifiers.
- A set of IRI references $V_{IRI}$ which is a subset of $V_{AID}$. $V_{IRI}$ consists of all full IRIs references and QNames.
- A set of concept identifiers $V_{C}$ which is a subset of $V_{ID}$. $V_{C}$ contains `wsml:true` and `wsml:false`.
- A set of datatype identifiers $V_{D}$ which is a subset of $V_{ID}$. $V_{D}$ consists of the XML Schema basic datatypes, `rdfs:Literal` and all user-defined datatypes.
- A set of identifiers for relations $V_{R}$ which is a subset of $V_{ID}$
- A set of identifiers for functions $V_{F}$ which is a subset of $V_{ID}$
- A set of attribute identifiers $V_{Att}$ which is a subset of $V_{RU}$
- A set of instance identifiers $V_{I}$ which is a subset of $V_{ID}$
- A set of literals $V_{L}$ which is a subset of $V_{ID}$
- A set of axiom identifiers $V_{A}$ which is a subset of $V_{AID}$
- A set of non-functional property identifiers $V_{NFP}$ which is a subset of $V_{ID}$
- A set of ontology identifiers $V_{O}$ which is a subset of $V_{AID}$
- A set of goal identifiers $V_{G}$ which is a subset of $V_{AID}$
- A set of web service identifiers $V_{WS}$ which is a subset of $V_{AID}$
- A set of capability identifiers $V_{Cap}$ which is a subset of $V_{AID}$
- A set of interface identifiers $V_{In}$ which is a subset of $V_{AID}$
- A set of choreography identifiers $V_{Cho}$ which is a subset of $V_{AID}$
- A set of orchestration identifiers $V_{Orc}$ which is a subset of $V_{AID}$
- A set of mediator identifiers $V_{M}$ which is a subset of $V_{AID}$
- A set of ooMediator identifiers $V_{OOM}$ which is a subset of $V_{M}$
- A set of ggMediator identifiers $V_{GGM}$ which is a subset of $V_{M}$
- A set of wgMediator identifiers $V_{WGM}$ which is a subset of $V_{M}$
- A set of wwMediator identifiers $V_{WWW}$ which is a subset of $V_{M}$

Single variants in the family of WSML languages can and do impose specific restrictions on the elements of a WSML vocabulary as well as relations among these elements in the context of the specific variant.
(c) Datatypes in WSML

The treatment of datatypes in WSML is inherited from WSMO [Roman et al., 2004], which is inherited from RDF. The recommended datatypes in WSML-Core are the XML Schema datatypes. In fact, any implementation of WSML is required to support the **xsd:string** and the **xsd:integer** datatypes. The datatype **numeric** indicates that an operator is not only applicable to **xsd:integer**, but also to any other numeric datatype (e.g. **xsd:float**, **xsd:decimal**) which is supported by the implementation. Furthermore, the following built-in predicates have to be supported by any WSML-compliant application:

<table>
<thead>
<tr>
<th>WSML datatype predicate</th>
<th>XQuery function</th>
<th>Datatype (A)</th>
<th>Datatype (B)</th>
<th>Return datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsml:numeric-equal(A,B)</td>
<td>op:numeric-equal(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml:numeric-greater-than(A,B)</td>
<td>op:numeric-greater-than(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml:numeric-less-than(A,B)</td>
<td>op:numeric-less-than(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml:string-equal(A,B)</td>
<td>op:numeric-equal(fn:compare(A,B), 1)</td>
<td>xsd:string</td>
<td>xsd:string</td>
<td>xsd:string</td>
</tr>
<tr>
<td>wsml:numeric-add(range,A,B)</td>
<td>op:numeric-add(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml:numeric-subtract(range,A,B)</td>
<td>op:numeric-subtract(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml:numeric-multiply(range,A,B)</td>
<td>op:numeric-multiply(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml:numeric-divide(range,A,B)</td>
<td>op:numeric-divide(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
</tbody>
</table>

These predicates correspond to functions in XQuery/XPath [Malhotra et al., 2004]. Notice that SWRL [Horrocks et al., 2004] built-ins support is also based on XQuery/XPath.

Each implementation is required to either implement the complement of each of there built-ins or to provide a negation operator which can be used together with these predicates.

In order to use a datatype, the datatype must be **known**, which means that datatype identifiers are known to refer to a datatype. All XML Schema built-in datatypes [Biron & Malhorta, 2004] are **known** datatypes.

II – Common elements of all WSML variants

This section describes the elements common between all WSML specifications and all WSML variants. The elements described in this section are used in ontology, goal, mediator and web service specifications. The elements specific to a type of specification are described in subsequent sections. Because all elements in this section are concerned with meta-information about the specification and thus do not depend on the logical
formalism underlying the language, these elements are shared among all WSML variants.

In this section we only describe how each element should be used. The subsequent sections will describe how these elements fit in the specific WSML descriptions.

A WSML document consists of the following:

\[
\text{wsml} = \text{wsmlvariant} \text{ namespace} \text{ definition}^* \\
\text{definition} = \{\text{goal}\} \text{ goal} \\
\]

\[
\text{wsmlvariant} = \text{wsmlvariant' full_iri} \\
\]

(a) WSML Variant

Every WSML specification document may start with the \text{wsmlvariant} keyword, followed by an identifier for the WSML variant which is used in the document. Table 2.1 lists the WSML variants and the corresponding identifiers in the form of IRIs.

WSML Variant IRI

Table 2.5.2: WSML variant identifiers

<table>
<thead>
<tr>
<th>WSML Variant</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSML-Core</td>
<td><a href="http://www.wsmo.org/2004/wsml/wsml-core">http://www.wsmo.org/2004/wsml/wsml-core</a></td>
</tr>
</tbody>
</table>

The specification of the \text{wsmlvariant} is optional. In case no variant is specified, no guarantees can be made with respect to the specification and WSML-Full may be assumed.

\[
\text{wsmlvariant} = \text{wsmlvariant' full_iri} \\
\]

The following illustrates the WSML variant reference for a WSML-Core specification:

\[
\text{wsmlvariant} <\text{http://www.wsmo.org/2004/wsml/wsml-flight}> \\
\]

By explicitly stating the intended WSML variant, tools can immediately recognize the intention of the author and return an exception in case the specification does not fall in the intended variant. This generally helps developers of WSML specifications to stay within desired limits of complexity and to communicate their desires to others.
(b) Namespace References

At the top of a WSML document, below the identification of the WSML variant, there is an optional block of namespace references, which is preceded by the namespace keyword. The namespace keyword is succeeded by a number of namespace references. Each namespace reference, except for the default namespace, consists of the chosen prefix and the IRI which identifies the namespace. Notice that, like any argument list in WSML, the list of namespace references is delimited with curly brackets ‘{‘ ‘}’.

\[
\text{namespace} = \text{'namespace'} \text{ prefixdefinitionlist}
\]
\[
\text{prefixdefinitionlist} = \{\text{defaultns}\} \text{ full_iri} \\
| \{\text{prefixdefinitionlist}\} \{' prefixdefinition moreprefixdefinitions* '}
\]
\[
\text{prefixdefinition} = \{\text{nspacedef}\} \text{ ncname full_iri} \\
| \{\text{default}\} \text{ full_iri}
\]
\[
\text{moreprefixdefinitions} = \{', \text{ prefixdefinition}\}
\]

An example:

\[
\text{namespace} \{<\text{http://www.example.org/ontologies/example#}>, \\
\text{dc} <\text{http://purl.org/dc/elements/1.1#}>, \\
\text{foaf} <\text{http://xmlns.com/foaf/0.1/}>, \\
\text{xsd} <\text{http://www.w3.org/2001/XMLSchema#}>, \\
\text{wsml} <\text{http://www.wsmo.org/2004/wsml#}>, \\
\text{loc} <\text{http://www.wsmo.org/ontologies/location#}>, \\
\text{oo} <\text{http://example.org/ooMediator#}>\}
\]

(c) Non-Functional Properties

Non-functional properties may be used for the WSML document as a whole but also for each element in the specification. Non-functional property blocks are identified by the keyword nonFunctionalProperties or nfp. Following the keyword is a list of attribute values, which consists of the attribute identifier, the keyword hasValue and the value for the attribute, which may be any identifier and can thus be an IRI, a literal or an anonymous identifier. The recommended properties are the properties of the Dublin Core [Weibel et al. 1998], but the list of properties is extensible and thus the user can choose to use properties coming from different sources. WSMO [Roman et al., 2004] defines a number of properties which are not in the Dublin Core. These properties can be used in a WSML specification by referring to the WSML namespace (http://www.wsmo.org/2004/wsml). These properties are:

wsml:security, wsml:transactional, wsml:trust (here we assume that the prefix wsml has been defined as referring to the WSML namespace). The recommended usage of these properties has been defined in [Roman et al., 2004]. Following the WSML convention, if a property has multiple values, these are separated by commas and the list of values is delimited by curly brackets.

\[
\begin{align*}
\text{nonFunctionalProperties} & = \text{'nfp'} \ \text{attributevalue}\ast \text{'endnfp'} \\
\text{header} & = \{\text{nfp} \mid \text{usesmediator} \mid \text{importsontology} \} \\
\text{nfp} & = \text{'nfp'} \ \text{attributevalue}\ast \text{'endnfp'} \\
\end{align*}
\]

An example:

\[
\begin{align*}
\text{dc:title} & \text{ hasValue } \text{ 'WSML example ontology' } \\
\text{dc:subject} & \text{ hasValue } \text{ 'family' } \\
\text{dc:description} & \text{ hasValue } \text{ 'fragments of a family ontology to provide WSML examples' } \\
\text{dc:contributor} & \text{ hasValue } \{<\text{http://homepage.uibk.ac.at/~c703240/foaf.rdf}>, \\
& \quad <\text{http://homepage.uibk.ac.at/~csaa5569/}>, \\
& \quad <\text{http://homepage.uibk.ac.at/~c703239/foaf.rdf}>, \\
& \quad <\text{http://homepage.uibk.ac.at/~c703319/foaf.rdf}>\} \\
\text{dc:date} & \text{ hasValue } \text{ '2004-11-22'\text{^xsd:date} } \\
\text{dc:format} & \text{ hasValue } \text{ 'text/html' } \\
\text{dc:language} & \text{ hasValue } \text{ 'en-US' } \\
\text{dc:rights} & \text{ hasValue } \text{ '<http://www.deri.org/privacy.html>' } \\
\text{wsml:version} & \text{ hasValue } \text{ '$Revision: 1.34$' } \\
\end{align*}
\]

\text{endNonFunctionalProperties}

(d) Importing Ontologies

Ontologies may be imported in any WSML specification through the imported ontologies block, identified by the keyword importsOntology. Following the keyword is a list of IRIs identifying the ontologies being imported. An importsOntology definition serves to merge ontologies, similar to the owl:import statement in OWL. This means the resulting ontology is the union of all axioms in the importing and imported ontologies. Note that also the default namespaces are merged; the default namespace of the importing ontology becomes also the default namespace of the resultant merged ontology. Please note that recursive import of ontologies is also supported. This means that if an imported ontology has any imported ontologies of its own, also these ontologies are imported.
**importsontology** = 'importsOntology' idlist

An example:

```plaintext
importsOntology {
  "http://www.wsmo.org/ontologies/location",
  "http://xmlns.com/foaf/0.1"
}
```

In case the imported ontology falls in a different WSML variant than the importing specification, the resulting ontology falls in the most expressive of the two variants. If the expressiveness of the variants is to some extent disjoint (e.g. when importing a WSML-DL ontology in a WSML-Rule specification), the resultant will fall in the least common superset of the variants. In the case of WSML-DL and WSML-Rule, the least common superset is WSML-Full.

### (d) Using Mediators

Mediators are used to link different WSML elements (ontologies, goal, web services) and resolve heterogeneity between the elements. Mediators are described in more detail in deliverable D16.1. Here, we are only concerned with how mediators can be referenced from a WSML specification.

The (optional) used mediators block is identified by the keywords `usesMediator` which is followed by one or more identifiers of WSMO mediators. The types of mediators that can be used are constrained by the type of specification. An ontology allows for the use of different mediators than, for example, a goal or a web service. The type of the mediator is reflected in the mediator specification itself and not in the reference to the mediator.

```plaintext
usesmediator = 'usesMediator' idlist
```

An example:

```plaintext
usesMediator "http://example.org/ooMediator"
```

### (e) Ontology specification in WSML

A WSML ontology specification is identified by the `ontology` keyword optionally followed by a IRI which serves as the identifier of the ontology. This identifier is used as the target namespace of the definitions in the ontology specification document. In case no identifier is specified for the ontology, the locator of the ontology serves as identifier.

An example:

```plaintext
ontology family
```
An ontology specification document in WSML consists of:

\[
\text{ontology} \quad = \quad '\text{ontology}' \text{ id?} \quad \text{header*} \quad \text{ontology_element*} \\
\text{ontology_element} \quad = \quad \{\text{concept}\} \quad \text{concept} \\
\{}\text{instance}\} \quad \text{instance} \\
\{}\text{relation}\} \quad \text{relation} \\
\{}\text{function}\} \quad \text{function} \\
\{}\text{relationinstance}\} \quad \text{relationinstance} \\
\{}\text{axiom}\} \quad \text{axiom}
\]

The ontology elements are described in more detail in the subsequent sections which describe the specific variants of WSML.

(f) Other elements of WSML specifications

So far, we have mentioned all elements of the WSML language(s) which are relevant for the description of ontologies and thus this deliverable within the Esperonto project. However, we want to stress that there are further elements of the language that we have not described here, since they are not relevant for our purpose here, namely, goal description, web service descriptions as well as mediator descriptions. Detail on the description of these elements can be found in the respective deliverable D16.1 of the WSML working group.

2.5.1.3 WSML-Core

WSML-Core is based on the semantics of OWL DL- [de Bruijn et al., 2004], which is based on the Logic Programming subset of Description Logics described in [Grosos et al., 2003]. Furthermore (and in contrast to former versions of WSML-Core that we discussed in previous versions of the deliverable), WSML-Core now uses a restricted form of the OWL-E datatype extension [Pan and Horrocks, 2004] of OWL. The relation between this extension and OWL DL- is described in [de Bruijn et al., 2004a]. The modeling constructs of WSML-Core are based on the conceptual model of WSMO [Roman et al., 2004]. Because WSML-Core is based on OWL DL- and there exists a translation from OWL DL- to plain (function- and negation-free) Datalog, the decidability and complexity results of Datalog apply to WSML-Core as well. The most important result is that Datalog is data complete for P, which means that query answering can be done in polynomal time.[2]

Most of the restrictions posed by WSML-Core are a consequence of the limitation of WSML-Core to the OWL semantics.

WSML-Core contains all common WSML elements described in the previous subsection. Furthermore, it describes a number of ontology modeling constructs and a language for logical expressions.

This chapter is structured as follows. We first introduce basics of the WSML-Core syntax, such as the use of namespaces, identifiers, etc. Then, we describe the modeling
of ontologies. Finally, we describe the logical expression syntax of WSML-Core which allows to define axioms within ontologies.

(a) Basic WSML-Core Syntax

WSML-Core inherits the basic of the WSML syntax specified in the previous section. In this section we describe restrictions WSML-Core poses on the basic syntax.

WSML-Core inherits the namespace mechanism of WSML.

WSML-Core restricts the use of the WSML vocabulary. The vocabulary of WSML-Core is separated similarly to OWL DL. More precisely, the following sets of identifiers are pairwise disjoint:

- The WSML-Core keywords
- The set of datatype identifiers
- The set of data-valued attribute and relation identifiers
- The set of general attribute and relation identifiers
- The set of concept identifiers

Definition 2. A WSML-Core vocabulary $V$ is a WSML vocabulary according to Definition 1 with the following additional restrictions:

- $V_{C}, V_{D}, V_{R}, V_{I}$ and $V_{NFP}$ are subsets of $V_{AID}$.
- $V_{Att} = V_{RU}$
- A set of relations with an abstract range $V_{RA}$ which is a subset of $V_{R}$.
- A set of relations with a concrete range $V_{RC}$ which is a subset of $V_{R}$.
- $V_{R} = V_{RA} \cup V_{RC}$
- $V_{RA}$ and $V_{RC}$ are disjoint
- $V_{C}, V_{D}, V_{R}, V_{I}$ and $V_{NFP}$ are pairwise disjoint

Note that attributes are special kinds of relations. Namely, binary relations with a defined domain. Therefore, it is possible to further define an attribute in a relation definition. These relation definitions would then affect the behavior of the attributes in reasoning. It is therefore generally not advisable to use the same identifier in both attribute and relation definitions.

(b) WSML-Core Ontologies

In this section we explain the ontology modeling elements in the WSML-Core language. The modeling elements are based on the WSMO conceptual model of ontologies [Roman et al., 2004].

Concepts

A concept definition starts with the concept keyword, which is optionally followed by the identifier of the concept. This is optionally followed by a superconcept definition which consists of the keyword subConceptOf followed by one or more concept identifiers (as usual, if there is more than one, the list is comma-separated and delimited by curly brackets). This is followed by an optional nonFunctionalProperties block
and zero or more attribute definitions. An attribute definition consists of the identifier of an attribute, the ofType or impliesType keyword and the identifier of a datatype (in the case of ofType) or a concept (in the case of impliesType) to which the attribute values must adhere. Note that the type of the attribute is optional.

\[
\begin{align*}
\text{concept} & = \ 'concept' \ \text{id} \ \text{superconcept}? \ \text{nfp}? \ \text{attribute}^* \\
\text{superconcept} & = \ 'subConceptOf' \ \text{idlist}
\end{align*}
\]

An example:

\[
\begin{align*}
\text{concept} \ \text{Human} & \ \text{subConceptOf} \ \{\text{Primate}, \ \text{LegalAgent}\} \\
\text{nonFunctionalProperties} & \\
\text{dc:description} \ \text{hasValue} & \ '\text{concept of a human being}' \\
\text{endNonFunctionalProperties} & \\
\text{hasName} & \ \text{impliesType} \ \text{foaf:name} \\
\text{hasParent} & \ \text{inverseOf} (\text{hasChild}) \ \text{impliesType} \ \text{Human} \\
\text{hasChild} & \ \text{impliesType} \ \text{Human} \\
\text{hasAncestor} & \ \text{transitive} \ \text{impliesType} \ \text{Human} \\
\text{hasWeight} & \ \text{ofType} \ \text{xsd:float} \\
\text{hasWeightInKG} & \ \text{ofType} \ \text{xsd:float} \\
\text{hasBirthdate} & \ \text{ofType} \ \text{xsd:date} \\
\text{hasObit} & \ \text{impliesType} \ \text{xsd:date} \\
\text{hasBirthplace} & \ \text{impliesType} \ \text{loc:location} \\
\text{isMarriedTo} & \ \text{symmetric} \ \text{impliesType} \ \text{Human} \\
\text{hasCitizenship} & \ \text{impliesType} \ \text{oo:country}
\end{align*}
\]

WSML-Core allows for a restricted form of logical expressions which can be used inside concept definitions in order to refine the definition which is already given by the subconcept and attribute definitions.

Different knowledge representation languages, such as Description Logics, allow for the specification of defined concepts (called "complete classes" in OWL). The definition of a defined concept is not only necessary, but also sufficient. For a necessary definition, such as the concept specification in the example above, specifies implications for all instances of this concept. Above, it specifies that each instance of Human is also an instance of Primate and LegalAgent. Furthermore, all values for the attributes hasName, hasParent, hasWeight etc. must be of specific types. A necessary and sufficient definition also works the other way around. For the example, every individual which is an instance of Primate and LegalAgent and which has specific types for all values of the attributes hasName, hasParent and hasWeight etc. is inferred to be an instance of Human.

WSML-Core supports defined concepts only to a limited extent. Namely, a concept can be defined by a conjunction of other concepts; attribute definitions are not allowed for defined concepts. WSML-Core does not provide additional keywords for defined
classes. Instead, the logical expression syntax can be used for defined classes. The logical expression should reflect an equivalence relation between a class membership expression on one side and a conjunction of class membership expressions on the other side, each with the same variable. Thus, such a definition should be of the form:

\[ ?x \text{memberOf } A \text{ equivalent } ?x \text{memberOf } B_1 \text{ and } ... \text{ and } ?x \text{memberOf } B_n \]

With \( A \) and \( B_1, ..., B_n \) concept identifiers.

For example, in order to define the class Human as the intersection of the classes Primate and LegalAgent, the following definition is used:

```xml
<concept Human
    nonFunctionalProperties
        dc:relation hasValue HumanDef
    endNonFunctionalProperties>

<axiom HumanDef
    definedBy
        ?x memberOf Human equivalent
        ?x memberOf Primate and ?x memberOf LegalAgent.
```

Attributes

WSML-Core allows two kinds of attribute definitions, namely datatype attribute definitions with the keyword `ofType` and concept attribute definitions with the keyword `impliesType`.

An attribute definition of the form \( A \text{ ofType } D \), where \( A \) is an attribute identifier and \( D \) is a datatype identifier, is a constraint on the values for attribute \( A \). If the value for the attribute \( A \) is not of type \( D \), the constraint is violated and the attribute value is inconsistent with respect to the ontology. This notion of constraints corresponds the usual database-style constraints and also the universal values restrictions for DatatypeProperties in OWL.

The keyword `impliesType` can be used for inferring the type of a particular attribute value. This type of attribute specification is only allowed for abstract types (i.e. concepts) and not for datatypes, because the semantics of WSML-Core does not allow for inferring the type of a literal. This restriction follows from the OWL semantics.

Concept attributes (i.e. attributes that do not have a datatype as range) can be specified as being transitive, symmetric, or being the inverse of another attribute, using the `transitive`, `symmetric` and `inverseOf` keywords, respectively. Notice that these keywords do not enforce a constraint on the attribute, but are used to infer additional information about the attribute. The keyword `inverseOf` must be followed by an identifier of the attribute, enclosed in parentheses, of which this attribute is the inverse.
\texttt{att\_type} = \{\text{constraint}\} 'ofType' \\
| \{\text{restriction}\} 'impliesType' \\
\texttt{attribute} = [\text{attr}]: \text{id} \ \text{attributefeature}\* \text{att}\ [\text{type}]: \text{id} \ \text{nfp}? \\
| \{\text{transitive}\} 'transitive' \\
\texttt{attributefeature} = | \{\text{symmetric}\} 'symmetric' \\
| \{\text{inverse}\} 'inverseOf' ('id') \\

When an attribute is specified as being transitive, this means that if three individuals $a$, $b$ and $c$ are related via a transitive attribute $att$ in such a way: $a\ att\ b\ att\ c$ then $c$ is also a value for the attribute $att$ at $a$: $a\ att\ c$.

When an attribute is specified as being symmetric, this means that if an individual $a$ has a symmetric attribute $att$ with value $b$, then $b$ also has attribute $att$ with value $a$.

When an attribute is specified as being the inverse of another attribute, this means that if an individual $a$ has an attribute $att1$ with value $b$ and $att1$ is the inverse of a certain attribute $att2$, then it is inferred that $b$ has an attribute $att2$ with value $a$.

\textbf{Relations}

A relation definition starts with the \texttt{relation} keyword, which is optionally followed by the identifier of the relation. This is optionally followed by a superrelation definition which consists of the keyword \texttt{subRelationOf} followed by one or more relation identifiers (as usual, if there is more than one, the list is comma-separated and delimited by curly brackets). This is followed by an optional \texttt{nonFunctionalProperties} block. In WSML-Core, relations are restricted to binary predicates, which correspond with ObjectProperties and DatatypeProperties in OWL.

\texttt{relation} = 'relation' \text{id} \text{superrelation}? \text{nfp}? \\
\texttt{superrelation} = 'subRelationOf' \text{idlist} \\

An example:

\texttt{relation hasAncestor subRelationOf hasRelative} \\
\texttt{nonFunctionalProperties} \\
\texttt{dc:description hasValue 'Relation between ancestors'} \\
\texttt{endNonFunctionalProperties} \\

\textbf{Functions}

WSML-Core allows the user to create functions, which are relations with a domain and a unary range. In WSML-Core, both the domain and the range must be datatypes. A function definitions starts with the \texttt{function} keyword, followed by the name (identifier) of the function. This is followed by an optional \texttt{nonFunctionalProperties} block.
The example below defines a new predicate for calculating the age of a human. The computation is done by use of in-built predicates.

An example of a function:

```xml
function AgeOfHuman

nonFunctionalProperties
  dc:description hasValue 'Function calculating the age of a human given its birthdate.'
  dc:relation hasValue AgeOfHumanDef
endNonFunctionalProperties

axiom AgeOfHumanDef
  definedBy
    forAll ?x,?y,?z ( AgeOfHuman(human hasValue ?x, range hasValue ?y) equivalent
      ?x memberOf Human and
      wsml:years-from-duration(?y,?z) and
      wsml:subtract-dateTimes-yielding-dayTimeDuration( ?z,
      ?x.hasBirthdate, wsml:current-dateTime())
    ).
```

**Instances**

An instance definition starts with the `instance` keyword, (optionally) followed by the identifier of the instance, the `memberOf` keyword and the name of the concept to which the instance belongs. An instance corresponds with an individual in OWL DL-. The `memberOf` keyword identifies the concept to which the instance belongs. This definition is followed by the attribute values associated with the instance. Each property filler consists of the property identifier, the keyword `hasValue` and the value(s) for the attribute. If an attribute has a datatype as its range, the attribute value should be a (possibly typed) literal. Note that the first two literals in the example are typed. The first is of type `xsd:string`, as a literal written between single quotes (‘...’) is interpreted as a typed literal of type `xsd:string`. The second value is a literal of type `xsd:date`.

```xml
instance = 'instance' id memberof? nfp? attributevalue*
memberof = 'memberOf' idlist
attributevalue = id 'hasValue' idlist
```
An example:

```
instance Mary memberOf {Parent, Woman}

nfp
dc:description hasValue "Mary is parent of the twins Paul and Susan"^^xsd:string
endnfp
hasName hasValue 'Maria Smith'
hasBirthdate hasValue "1949-09-12"^^xsd:date
hasChild hasValue {Paul, Susan}
```

Instances explicitly specified in an ontology are those that are shared together with the ontology. However, most instance data exists outside the ontology in private data stores. In order to access these instances, as described in [Roman et al., 2004], is by providing a link to an instance store. Instance stores contain large numbers of instances and they are linked to the ontology. We do not restrict the user in the way an instance store is linked to a WSML-Core ontology. This would be done outside the ontology definition, since an ontology is shared and can thus be used in combination with different instance stores.

Besides specifying instances of concepts, it is also possible to specify instances of relations. Such a relation instance definition starts with the `relationInstance` keyword, (optionally) followed by the identifier of the relationInstance, the `memberOf` keyword and the name of the relation to which the instance belongs. This is followed by an optional `nonFunctionalProperties` block, followed by the values of the parameters associated with the instance. Each parameter value consists of the parameter identifier, the keyword `hasValue` and the value for the parameter (notice that a parameter may only have one value).

An example:

```
relationInstance ageJohn hasAge(John,23)
```

Axioms

An axiom definition starts with the `axiom` keyword, followed by the name (identifier) of the axiom. This is followed by an optional `nonFunctionalProperties` block and a logical expression preceded by the `definedBy` keyword. The language allowed for the logical expression is explained in Section 3.6.

```
axiom = 'axiom' axiomdefinition

axiomdefinition = {use_axiom} id
                | {defined_axiom} id? nfp? log_definition
```
\texttt{log\_definition} = 'definedBy' \texttt{log\_expr}

An example of a defining axiom:

\begin{verbatim}
axiom humanDefinition
definedBy
    ?x memberOf Human equivalent
    ?x memberOf Animal and
    ?x memberOf LegalAgent.
\end{verbatim}

WSML-Core allows to specify constraining axiom for datatype predicates. An example of a constraining axiom:

\begin{verbatim}
axiom humanBMIConstraint
definedBy
    false impliedBy
    naf bodyMassIndex(range hasValue ?b, length hasValue ?l, weight hasValue ?w) and
    ?x memberOf Human and
    and
    ?x[length hasValue ?l, weight hasValue ?w, bmi hasValue ?b].
\end{verbatim}

In the example we use default negation of the user-defined datatype predicate \texttt{bodyMassIndex}. In case default negation is not supported by the reasoner, the complement of the user-defined predicate needs to be defined in order for use in constraints.

\textit{(c) WSML-Core Logical Expression Syntax}

In this paragraph we explain the basic syntax of logical expressions used in the WSML-Core variant. All other variants define extensions of this syntax.

Logical expressions may be simple or complex. A logical expression is terminated by a period. Simple logical expressions can be combined to form complex expressions.

WSML has the following simple logical expressions:

- Relation Expressions
- Molecules

WSML has the following complex logical expressions:

- Compound Logical Expressions
- Formulas

The definition of the logical expression syntax is a restricted form of the logical expression syntax defined in [Roman et al., 2004, Chapter 7], with a few modifications.
Instead of describing the exact restrictions and modifications, we will, in future versions, provide here the complete definition of the WSML-Core basic logical expression syntax.

An important aspect of logical expressions in WSML-Core, besides the severe limitations on the kinds of logical expressions that can be written, is the fact that all relations and attributes used in the logical expressions must be explicitly declared in the conceptual syntax. This is necessary in order to determine whether a relation or an attribute is a relation over the abstract domain or a relation over the abstract and the concrete domain.

Simple Logical Expressions

The two basic types of simple logical expressions are relation expressions, molecules and datatype predicates.

Besides these basic types of logical expressions, we allow the use of the keywords true and false. These keywords stand for universal truth and universal falsehood, respectively. true is equivalent to the formula $A \lor \neg A$ where $A$ is an arbitrary predicate and $\neg$ denotes classical negation. false is equivalent to the formula $A \land \neg A$ where $A$ is an arbitrary predicate. Therefore, true is in every model and false is not included in any of the models of a logical theory.

Relation Expressions

A relation logical expression consists of a predicate identifier followed by the comma-separated arguments of the predicate, enclosed by parentheses. A relation value logical expression is one that can be expressed by a single binary relation relating the two arguments. Both arguments can either be data values (i.e. literals) or identifiers referring to instances. However, if the first argument is a data value, the second argument must also be a data value. In the latter case, the relation actually corresponds to a datatype predicate.

An example: ageInYears(john, 23)

Molecules

A molecule in WSML-Core is either a concept molecule or an instance molecule.

An instance molecule is one of the following statements:

- A concept membership assertion of the form $I \texttt{memberOf} C$, where $I$ is an instance identifier and $C$ is a concept identifier.
- An attribute value list of the form $I[A_1 \texttt{hasValue} v_1, \ldots, A_n \texttt{hasValue} v_n]$, where $I$ is an instance identifier, $A_1, \ldots, A_n$ are attribute identifiers and $v_1, \ldots, v_n$ are either instance identifiers or data values.

A concept membership assertion of the form $I \texttt{memberOf} C$ is true iff $I$ is an instance of concept $C$. An attribute value list specifies the values for certain attributes for this particular instance.
Two examples:

?x memberOf Boy
?x[hasName hasValue 'George', hasParent hasValue Paul]

A concept molecule is one of the following statements:

- A subconcept assertion of the form $C \text{ subConceptOf } D$, where $C$ and $D$ are concept identifiers.
- An attribute definition list of the form $C[A_1 \text{ ofType } D_1, ..., A_i \text{ ofType } D_i, A_{i+1} \text{ impliesType } D_{i+1}, ..., A_n \text{ impliesType } D_n]$, where $I$ is an instance identifier, $A_1, ..., A_n$ are attribute identifiers, $D_1, ..., D_i$ are datatype identifier, and $D_{i+1}, ..., D_n$ are concept identifiers.

A subconcept assertion of the form $C \text{ subConceptOf } D$ is true iff $C$ is a subconcept of $D$, which means that each instance of $C$ is also an instance of $D$. An attribute definition of the form $C[A \text{ ofType } D]$ is true iff for each instance of concept $C$, each value for the attribute $A$ is of the type $D$.

Two examples:

Boy subConceptOf Man
Boy[hasName ofType foaf:name, hasChild impliesType Human]

Complex Logical Expressions

WSML-Core has the following complex logical expressions:

- **Compound Logical Expressions** consist of several molecules and/or (datatype) predicates, separated by the and and the or keywords. Notice that and binds stronger than or and thus has precedence. It is also possible to use parentheses '(' ')' in order to influence the precedence of certain constructs. We furthermore define conjunctive compound logical expressions:
  - A **Conjunctive Compound Logical Expression** is collection of molecules and/or (datatype) predicates, separated by the and keyword.

- **Formulas** consist of a logical expression, an implication symbol, and a logical expression. Both logical expressions can be either a simple or a compound logical expression.

Compound Logical Expressions

A compound logical expression consists of a number of simple logical expressions connected with the keyword and. The compound logical expression is satisfied if each of the simple logical expressions is satisfied.

An example:

Boy subConceptOf Man
Molecules involving the same instance or concept, occurring in a compound logical expression, can be collapsed into one compound molecule to allow for more concise syntax. The example above can be thus rewritten:

Boy[ofName ofType foaf:name, hasChild impliesType Human] subConceptOf Man

Formulas

A formula in WSML-Core consists of two (simple or compound) logical expressions, separated by an implication symbol. This implication symbol can be left implication (impliedBy), right implication (implies) or dual implication (equivalent). In formulas, variables are allowed to occur in the place of identifiers.

- Left implication: $E_1$ impliedBy $E_3$, where $E_1$ and $E_3$ are (simple or compound) logical expressions. $E_1$ is true wrt. a certain variable binding, if $E_3$ is true wrt. the same variable binding. $E_3$ is called the antecedent and $E_1$ is called the consequent of the formula.

- Right implication: $E_1$ implies $E_3$, where $E_1$ and $E_3$ are (simple or compound) logical expressions. $E_3$ is true wrt. a certain variable binding, if $E_1$ is true wrt. the same variable binding. $E_1$ is called the antecedent and $E_3$ is called the consequent of the formula.

- Dual implication: $E_1$ equivalent $E_3$, where $E_1$ and $E_3$ are (simple or compound) logical expressions. $E_1$ is true wrt. a certain variable binding if and only if $E_3$ is true wrt. the same variable binding. A dual implication $E_1$ <-> $E_3$ is actually equivalent to the two implications: $E_1$ <- $E_3$ and $E_1$ -> $E_3$. Therefore, both $E_1$ and $E_3$ are both the antecedent and the consequent of the formula.

Note that variables occurring in the consequent of a formula must also occur in the antecedent of the same formula. Note also that all variables are implicitly universally quantified outside of the formula.

An example:

?x memberOf Human and ?x.hasAgeInYears <= 14 implies ?x memberOf Child

In order to model a database-style integrity constraint in the logical language, the keyword false is required. An integrity constraint is modeled as a logical implication with false in the consequent, for example:

false impliedBy ?x memberOf Child and ?x.hasAgeInYears > 14.

This integrity constraint is violated if there is any child with an age superior to 14.
Datatype Expressions

Datatype Predicates

A datatype predicate consists of a predicate identifier followed by the comma-separated arguments of the predicate, enclosed by parentheses. The number of arguments depends on the arity of the predicate. Each argument must be a data value. Thus, datatype predicates are a special kinds of relations, namely relations with only datatypes as arguments. A datatype predicate can either be a built-in predicate (i.e. the extension is computed through some procedure outside the logical language) or constructed from built-in predicates through a datatype expression (see below the next section).

An example: \( \text{wsml:numeric-equals}(3,4) \)

The user is free to choose the built-in predicates as long as the implementation knows how to handle the built-ins (similar to the allowed datatypes). However, we recommend a minimal list of supported datatype predicates, which are listed in Table 2.5.1. These predicates operate on the domains of xsd:string and xsd:integer, which are the basic datatypes, which should be supported by any WSML-Core implementation.

WSML-Core allows infix notation for certain functions and certain relations, which are equivalent to corresponding datatype predicates. More specifically, we allow infix notation for the following built-in functions numeric addition ('+'), subtraction ('-'), multiplication ('*') and division ('/'). We allow infix notation for the following built-in relations: numeric and string equality ('='), numeric and string inequality ('!='), and the following numeric comparisons: greater than ('>'), less than ('<'), greater or equal ('>=') and less or equal ('=<'). See appendix D.2 for a list of syntactic shortcuts and a translation to datatype predicates.

Conclusions: WSML-Core

In this section we have introduced WSML-Core, which is the core variant of WSML, based on the intersection of Datalog and Description Logics.

The major limitations of WSML-Core are the limited use of logical expressions, the lack of negation, the lack of value constraints for attributes with concept ranges, the lack of cardinality constraints and the lack of n-ary relations. WSML-Flight, to specified in the next chapter, overcomes many of the limitations of WSML-Core.

2.5.1.4 WSML-Flight

WSML-Core has some limitations with respect to conceptual modeling. It is not possible to specify constraints on attributes with a concept as range and also cardinality constraints are not supported. Furthermore, WSML-Core does not support n-ary relations and has limitations on the logical expression syntax. Finally, WSML-Core has
no negation and allows only negative conclusions about datatype expressions. In order to overcome these limitations, we present WSML-Flight in this chapter.

WSML-Flight is both syntactically and semantically completely layered on top of WSML-Core. This means that every valid WSML-Core specification is also a valid WSML-Flight specification. Furthermore, all consequences inferred from a WSML-Core specification are also valid consequences of the same specification in WSML-Flight. Finally, if a WSML-Flight specification falls inside the WSML-Core fragment then all consequences wrt. the WSML-Flight semantics also hold wrt. the WSML-Core semantics.

The features added by WSML-Flight are the following:

- Allows n-ary relations with arbitrary parameters
- Attribute definitions for the abstract domain
- Cardinality constraints
- (Stratified) default negation in logical expressions (in the antecedent of the rule)
- (in)equality in the logical language (in the antecedent of the rule)
- Fully-fledged rule language
- No longer a separation of vocabulary (wrt. concepts, instances, relations); thus, WSML-Flight allows for meta-modeling

(a) Basic WSML-Flight Syntax

WSML-Flight adheres to the WSML syntax basics described in Section 2.5.1.2. The restrictions posed on this basic syntax by WSML-Core do not apply to WSML-Flight. The WSML Flight vocabulary is the same as the WSML vocabulary introduced in Definition 1.

(b) WSML-Flight Ontologies

The modeling elements of WSML-Flight ontologies are inherited from WSML-Core, although WSML-Flight does allow additional functionality for attribute definitions, relations, functions, and relation instances. In this section we only describe functionality added by WSML-Flight on top of WSML-Core.

Attributes

In WSML-Flight, the keyword ofType can be used for both regular attribute definitions and datatype attribute definitions. As in WSML-Core impliesType may only be used for concept attributes.

Besides the more general use of the ofType keyword, WSML-Flight adds two distinct features to attribute definitions compared with WSML-Core. The first feature is reflexivity. Reflexivity is asserted by using the reflexive keyword. The second feature is the cardinality constraints. The cardinality constraints for a single attribute are specified by including two numbers between parentheses '( ' )', indicating the minimal and maximal cardinality, after the ofType keyword. The first number indicates the minimal cardinality. The second number indicates the maximal cardinality, where '*' stands for
unlimited maximal cardinality (and is not allowed for minimal cardinality). It is possible to write down just one number instead of two, which is interpreted as both a minimal and a maximal cardinality constraint. When the cardinality is omitted, then it is assumed that there are no constraints on the cardinality, which is equivalent to (0 *). Notice that a maximal cardinality of 1 makes an attribute functional.

\[
\text{attribute} = [\text{attr}]: \text{id} \ \text{attributefeature}^* \ \text{att_type} \ \text{cardinality}\ ? [\text{type}]: \text{id} \ nfp?
\]

\[
\text{cardinality} = '\text{'} [\text{min_cardinality}]: \text{pos_int} \ [\text{max_cardinality}]: \text{cardinality_number} ? '\text{'}
\]

\[
\text{cardinality_number} = \{\text{finite_cardinality}\} \ \text{pos_int}
\]
\[
| \{\text{infinite_cardinality}\} '\text{*}'
\]

\[
\text{attributefeature} = \{\text{transitive}\} '\text{transitive}'
\]
\[
| \{\text{symmetric}\} '\text{symmetric}'
\]
\[
| \{\text{inverse}\} '\text{inverseOf}(\text{' id '}')
\]
\[
| \{\text{reflexive}\} '\text{reflexive}'
\]

An example of a concept definition with WSML-Flight attribute definitions:

```xml
<concept name="Human">
  <nonFunctionalProperties>
    <dc:description>hasValue 'concept of a human being'</dc:description>
  </nonFunctionalProperties>
  <hasName ofType="foaf:name"/>
  <hasParent inverseOf="hasChild">impliesType Human</hasParent>
  <hasChild impliesType="Human"/>
  <hasAncestor transitive impliesType="Human"/>
  <hasWeight ofType="xsd:float"/>
  <hasWeightInKG ofType="xsd:float"/>
  <hasBirthdate ofType="xsd:date"/>
  <hasObit ofType="(0 1) xsd:date"/>
  <hasBirthplace ofType="loc:location"/>
  <isMarriedTo symmetric impliesType="(0 1) Human"/>
  <hasCitizenship ofType="oo:country"/>
</concept>
```

**Relations**

Relations in WSML-Flight can have an arbitrary arity.

```
\[
\text{relation} = 'relation' \ id \ narity? \ superrelation? \ nfp?
\]
```
superrelation = 'subRelationOf' idlist

arity = '/\ pos_int

An example:

```
relation distance/3 subRelationOf measurement
```

**Functions**

function = 'function' id arity superrelation nfp?

arity = '/\ pos_int

An example:

```
function AgeOfHuman/2
nonFunctionalProperties
  dc:description hasValue 'Function calculating the age of a human
given its birthdate.'
  dc:relation hasValue DefAgeOfHuman
endNonFunctionalProperties

axiom DefAgeOfHuman
  definedBy
    forAll ?x,?y,?z ( 
      AgeOfHuman(human hasValue ?x, range hasValue ?y) equivalent
      ?x memberOf Human
      and wsml:yearsFromDuration(?y, ?z)
      and wsml:subtractDateTimesYieldingDayTimeDuration(
        ?z, ?x.hasBirthdate, 
        wsml:currentDateTime())
    ).
```

**Relation Instances**

Because WSML-Flight allows relations of arbitrary arity, thus also the relation instances in WSML-Flight are slightly different from WSML-Core.
An example of an instance of a ternary relation (remember that the identifier is optional):

\[ \text{relationInstance distance (Innsbruck, Munich, 234)} \]

(c) WSML-Flight Logical Expression Syntax

The language for specifying logical expressions in WSML is not fully defined yet. Most likely it will be based on Datalog extended by stratified negation.

(d) Differences between WSML-Core and WSML-Flight

The features added by WSML-Flight compared with WSML-Core are the following:

- Allows n-ary relations with arbitrary parameters
- Attribute definitions for the abstract domain
- Cardinality constraints
- (Stratified) default negation in logical expressions
- (in)equality in the logical language (in the body of the rule)
- Full-fledged rule language
- No longer a separation of vocabulary (wrt. concepts, instances, relations); thus, WSML-Flight allows for meta-modeling, i.e. the same identifier can done both a class, an instance, an attribute, a relation, etc.

2.5.1.5 Summary

The Web Service Modeling Language (WSML) represents a family of representation languages for Ontologies as well as Semantic Web Services.

The single variants of the language share a common base syntax which can be restricted and extended as it is required for the single languages.

At present, only two of the variants in the language are specified, namely WSML-Core and WSML-Flight, whereby for WSML-Flight the language for defining logical expressions is not defined yet.

We presented the part of WSML and the two already defined WSML-variants that are relevant for the description of ontologies and the Esperonto project.

Recently, the WSML working group started some efforts on the implementation of reasoners for the single languages. But so far, there is no dedicated system available which could readily be used for applications which are based on WSML-Ontologies.

Resources

http://www.wsmo.org/2004/d16/d16.1/v0.2/20041224/
2.6 Ontology language in the DIP project

Overview

The DIP project (Data, Information, and Process Integration with Semantic Web Services)

DIP’s primary objective is to develop and extend Semantic Web and Web Service technologies in order to produce a new technology infrastructure for Semantic Web Services (SWS) - an environment in which different web services can discover and cooperate with each other automatically. DIP’s long term vision is to deliver the potential benefits of Semantic Web Services to e-Work and e-Commerce.

Workpackage WP2 of DIP is concerned with the development of an ontology management tools suite for the project. One of the deliverables in the workpackage is an ontology representation language which is scheduled for delivery in June 2005.

The ontology language to be developed in the course of the DIP project will address some really interesting aspects in regard of large-scale and advanced applications in an open and dynamic environment:

- Large-scale ontologies
- Heterogeneous ontologies and ontology networks
- Dynamic ontologies (i.e. ontologies that are evolving over time)
- Increased expressivity for advanced applications

The work on this language has started only recently, such that there are no official or informal documents available. Based on private communication with people involved we can basically outline the plans for addressing these requirement as follows:

- **Large-scale ontologies**: Include some means for modularizing ontologies into smaller pieces, support for networks of ontologies, support for the integration of large sets of instance data stored on external storage devices.

- **Heterogeneous ontologies and ontology networks**: Support for the definition of mappings between heterogeneous ontologies in an ontology network and ontology mediation.

- Dynamic ontologies (i.e. ontologies that are evolving over time): Some support for versioning of ontologies and their constituent elements.

- **Increased expressivity for advanced applications**: Extension of the ontology language by means of a rule language (in a decidable manner)

2.5.3 Summary

The design and development of the language has just started and there is no information publicly available at present. A first version of the language will be available (within the DIP project) in January 2005. This version of the language most likely will support only some of the features mentioned above.

**Resources**

http://dip.semanticweb.org
2.6.1 Updates and improvements: August 2004-January 2005

The DIP project eventually decided to adopt and use the WSML family of languages for the description of Ontologies as well as the Semantic Description of Web Services. Hence, all the comments and updates mentioned in Section 2.5.1 apply for the DIP project from now on as well.

It can be expected that the DIP project will play an important role for the evaluation and evolution of the WSML family of languages, since it provides an industrial-based test bed with concrete use cases that have to be described and tackled by the WSML languages.
3. Choosing an Ontology Language

This section presents a brief summary of pros and cons of the different Semantic Web languages presented in D2.1 and overviewed in this deliverable, in order to try to help the decision making process of which language to use in the Esperonto project.

3.1 Requirements of the Esperonto project

The nature and kind of work of the Esperonto project requires some functionality which in the case of being included in the Semantic language to be chosen would make the development easier.

As in any other Semantic Web project, Esperonto requires a degree of balance between expressive power and reasoning support, being the most important functionalities for the project. Expressive power will decrease very much the efforts towards the formalization of the Esperonto knowledge domains, providing a rich set of primitives that allow a wide variety of knowledge to be formalized. Reasoning capabilities should be powerful enough as to deal with the expressed knowledge in a fast and efficient way. Scalability, due to agent based approach of the project, is also a must that the reasoning mechanisms must be powerful enough to hold.

Another important aspect that should be taken under consideration when making the final language decision is the equality and inequality support. Due to the agent based approach of the project, it is necessary to align the different vocabularies used by agents to a common terminology that can be understood and shared among all of them. From this point of view, such facilities are of special interest since they will ease in great manner the communication efforts carried by different agents.

Some other requirements that complement this list are:

1. **Ease of use.** Due to actual lack of tools and that the project is an early adopter of the language, it should have a human readable syntax and fast learning curve.

2. **Compatibility.** There exists the possibility of using some other languages such as RDF(s) and OWL in some parts of the project. In order to minimize the efforts towards the final consolidation into a single language, compatibility with other standards should be supported by the chosen language.

3. **Internationalization.** Due to the multilinguality support Esperonto must offer (English, Spanish, Catalan) ontologies will be shared among different speaking agents. Due to this fact internationalization must be supported by the language.

4. **Sharing and versioning.** Ontologies will be shared and will evolve as the project evolves. If the language counts with support for these features, then the amount of work within the project related to these matters will be reduced.

5. **Simple extendibility to support advanced application.** Ontologies by themselves are static components of Semantic Web, and they are not useful if they can not be processed by applications, software agents and Web Services. Thus, they and their instance data serve as an input to process to applications that can perform certain actions on the basis of it. Hence, tracking if the ontology language chosen for the project is developed sufficiently enough to support advanced operations on ontologies specified in this language is an important issue. One of the most important and widely recognized aspects here is extended expressivity of the
languages for advanced applications like Semantic Web Service applications by means of rule support. Rules represent a very flexible mechanism to add domain knowledge to ontologies and derive implicit knowledge from the given facts in an elegant and efficient way. The rule extendibility of the candidate ontology languages is analyzed in section 3.3 of this deliverable.

In the next version of this deliverable we will include experience gained with the case studies to formulate more precisely the requirements of a project akin to Esperonto on a Semantic Web language.

3.2 Candidate languages

In this section the main candidates to be used within the project are presented, their strength and weaknesses are discussed, according to the project necessities, and taking special care of providing the different workarounds for all of them.

3.2.1 RDF(s)

This language lacks of sharing and versioning capabilities. Its expressive power is quite limited and the reasoning capabilities are not the strongest among the different languages, providing a limited reasoning mechanism only suitable for constraint checking. There are many tools and examples that could be either used or followed to learn about the language which makes it very widespread. Regarding internationalization, it supports different natural languages and it is compatible with XML, of which it is considered to be a super set. The community is active developing and improving this language. The language is an attractive choice for the project in the short term if its expressive power is considered being enough, apart from this it has limited reasoning capabilities. In the long run, the impact and visibility of the project can be compromised, if this is the chosen language. However, RDF(S) has been the language selected for the first prototype of the Esperonto system and of the case studies. Furthermore, a major argument for using RDF(S) is that RDF as seen as almost non-alternative languages for representing instances, since this is the language used by RDFS and OWL for expressing instances.

3.2.2 OIL

OIL has more of expressive power than RDF(S), which can make a better choice than the former. Though, OIL does not have versioning capabilities which may complicate application maintenance. The reasoning capabilities of OIL provide atomic consistency checking and allow cross linking the inter-ontology relations and checking for implied relations. Regarding interoperability, it allows the partial definition of mapping rules. It incorporates internationalisation facilities supporting different natural languages. It is quite easy to use and there is a lot of documentation and examples on the language use, as well as tools and support. The drawback of this language is that it is no longer under development, so the impact and visibility of the project could be limited.

3.2.3 DAML+OIL

Its reasoning capabilities are useful for ontology sharing. Just like OIL, it does not have a versioning support. Regarding interoperability, it allows the partial definition of mapping rules. Reasoning in DAML+OIL is specially suited for DL reasoning supporting design maintenance and deployment of ontologies. Its expressive power is much richer than the one of its predecessors, it supports different natural languages, it is quite easy to use, and regarding its compatibility it is important to make notice that it supports the full range of XML Schema datatypes since it is based on the existing Web standards (XML and RDF). This is also a language that is no longer under development compromising the impact and long term visibility of the project, just like OIL.
3.2.4 OWL

The reasoning functionalities of OWL could be used like in the case of DAML+OIL to provide sharing capabilities. Unlike the languages presented so far, it counts with versioning functionalities built in the language. Its reasoning mechanism is the same as in DAML+OIL and it is based on open world assumptions (OWA). It is equipped with a rich expressive power and possesses a layered architecture for scalability. The ease of use is a common feature of all the languages presented so far. OWL supports different natural languages as the rest of its colleagues.

OWL is very well positioned in the community mobilizing lots of efforts to make it become “the” Semantic Web language to use in the future. Recently the W3C has released 6 different recommendations for this ontology language, proving the efforts, support and expectative the community has in this language. Among them: (1) a presentation of the use cases and requirements that motivated OWL; (2) an overview document which briefly explains the features of OWL and how they can be used; (3) a comprehensive Guide that walks through the features of OWL with many examples of the use of OWL features; (4) a reference document that provides the details of every OWL feature; (5) a test case document, and test suite, providing over a hundred tests that can be used for making sure that OWL implementations are consistent with the language design; (6) a document presenting the semantics of OWL and details of the mapping from OWL to RDF.

From an impact and visibility point of view, this would be the language to chose, even though it could take some more effort to with it, especially in the development phase. At the moment there are many initiatives developing tools and documentation. Among them, the most interesting for Esperonto could be the one carried on by NetworksInference26, who are on their way to provide the first commercial OWL based inference engine basing on the expertise acquired in developing research laboratory reasoners as FaCT and RACER. Pellet27 at the Maryland Information and Network Dynamics Lab are other organizations that are on their way to develop an OWL reasoner.

As presented earlier in the document OWL is divided in three sub-languages suitable for different purposes. When choosing a particular language to work with, a comparison among the OWL sublanguage layers is necessary:

- **OWL-Lite vs OWL DL**
  - Depends upon the grade of expressiveness that needs to be achieved
  - Definition on RDF graph form
- **OWL DL vs OWL Full**
  - Whether the meta-modeling capabilities of RDF are required or not.
  - OWL Full can be viewed as an RDF extension
- **OWL Full vs OWL DL**
  - Reasoning support is predictable since there is no complete OWL full implementations

3.2.5 Ontology languages in the WSMO initiative

At present, there is only a precise definition and discussion for OWL-Lite-, a proper subset of OWL Lite with nice computational and extendability properties: There exists a direct translation into the deductive database language Datalog. Thus, any OWL Lite- ontology can be translated into Datalog in order to allow for efficient query answering. It turns out that most current ontologies fall inside this fragment. An ontology language for which a translation to Datalog

---

26 www.networkinference.com
exists has several advantages. Most notably, it can benefit from highly optimized query answering engines and allows for easy implementation of a rule and a query language on top of the ontology. Since OWL Lite- is a subset of OWL Lite, all tools that can deal with OWL Lite- can automatically be used for OWL Lite-. Moreover, by implementing the translation from OWL Lite- to Datalog, one can use any existing Datalog engine for efficient reasoning on ontologies and large sets of instances.

OWL Flight represents an extension of OWL Lite- with datatype support, Unique Name Assumption, Constraints, Classes-as-instances (or Metaclasses, resp.), and Local-closed world Assumption. Currently, the language is not fully defined in all aspects and there are no reasoning tools available.

OWL DL- and OWL Full- are intended extensions of OWL with attractive computational properties. At present there is no publicly available documentation and definition of the language. The next version of this deliverable will overview and discuss the state-of-affairs of the OWL- family of languages in the WSMO project as well as the available reasoning tools.

WSML-Core is a separate language that is based on OWL-Lite- and combines OWL-Lite- with the ontology model described in the WSMO ontology. In principle, this adds additional expressivity to OWL-Lite-. Currently, there is no implementation of reasoning support for WSML-Core, since the language is still in the process of being developed.

### 3.2.6 Ontology language of the DIP project

The ontology language to be developed in the course of the DIP project will address some really interesting aspects with regards to large-scale applications in an open and dynamic environment:

- Large-scale ontologies
- Heterogeneous ontologies and ontology networks
- Dynamic ontologies (i.e. ontologies that are evolving over time)
- Increased expressivity for advanced applications

The design and development of the language has just started and there is no information publicly available at present. A first version of the language will be available (within the DIP project) in January 2005. This version of the language most likely will support only some of the features mentioned above. We plan to include more material on the DIP ontology language in the next version of the deliverable. The ontology language of the DIP project can not be really considered to be the candidate language for Esperonto project, since it will not be released during the execution of the Esperonto project. However, it will be interesting to see whether integration with the DIP language could be good for next improvements of the project software.

### 3.3 Rule extensions of candidate languages

**Motivation.** Rules are considered to be a design issue for the Semantic Web (on top of the ontology layer in Tim Berners-Lee’s Semantic Web layer cake) and have been a topic of discussion in the W3C Web Ontology Working Group, but have not been included in the Web ontology language OWL. It is expected that there will be a W3C Working Group for developing a W3C rule markup language, possibly having a RuleML working group as a core.

Rule markup languages, that allow to express business rules as modular, stand-alone units in a declarative way, and to publish them and interchange them between different systems and tools,
will play an important role for facilitating business-to-customer (B2C) and business-to-business (B2B) interactions over the Web.

In a narrow sense, a rule markup language is a concrete (XML-based) rule syntax for the Web. In a broader sense, it should have an abstract syntax as a common basis for defining various concrete sublanguages serving different purposes.

Rules may be considered at three different abstraction levels:

1. At the business domain level, rules are statements that express (certain parts of) a business/domain policy (e.g., defining terms of the domain language or defining/constraining domain operations) in a declarative manner, typically using a natural language or a visual language. Examples are:
   - R1 “The driver of a rental car must be at least 25 years old”
   - R2 “A gold customer is a customer with more than $1 Million on deposit”
   - R3 “An investment is exempt from tax on profit if the stocks have been bought more than a year ago”
   - R4 “When a share price drops by more than 5% and the investment is exempt from tax on profit, then sell it”
   R1 is an integrity rule, R2 and R3 are derivation rules, and R4 is a reaction rule (see below for explanations of these rule categories). These appear to be the major semantic categories of business rules. Actually, many business rules appear to be reaction rules, which specify policies for real-world business behavior.

2. At the platform-independent level, rules are formal statements, expressed in some formalism or computational paradigm, which can be directly mapped to executable statements of a software platform. Rule languages used at this level are SQL:1999, OCL 2.0, and ISO Prolog. Remarkably, SQL provides operational constructs for all three business rule categories mentioned above: checks/assertions operationalize a notion of constraint rules, views operationalize a notion of derivation rules, and triggers operationalize a notion of reaction rules.

3. At the platform-specific level, rules are statements in a specific executable language, such as Oracle 10g views, Jess 3.4, XSB 2.6 Prolog, or the Microsoft Outlook 6 Rule Wizard.

Generally, rules are self-contained knowledge units that involve some form of reasoning. They may, for instance, specify

- static or dynamic integrity constraints (e.g. for constraining the state space or the execution histories of a system),
- derivations (e.g. for defining derived concepts),
- reactions (for specifying the reactive behavior of a system in response to events)

Given the linguistic richness and the complex dynamics of business domains, it should be clear that any specific mathematical account of rules, such as classical logic Horn clauses, must be viewed as a limited descriptive theory that captures just a certain fragment of the entire conceptual space of rules, and not as the only definitive, normative account. Therefore, in RuleML, a family of rule languages capturing the most important types of rules is being defined.
With respect to the requirement no. 5 specified in the section 3.2, this section contains analysis of state of the art for the rule extensions for some of the Semantic Web languages described in this deliverable.

Naturally, these languages interpret and exploit rules as *derivations* in the categorization mentioned above.

### 3.3.1 RDF/S rule extensions

**RDF extension: TRIPLE**

**Overview**

TRIPLE [Sintek and Decker, 2001], [Sintek and Decker, 2002] is a rule language for the Semantic Web. TRIPLE is based on Horn logic and borrows many basic features from F-Logic but is especially designed for querying and transforming RDF models. In this respect, it can be seen as a query and inference language for RDF. TRIPLE can be viewed as a successor of SiLRI (Simple Logic-based RDF Interpreter). One of the most important differences to F-Logic and SiLRI is that TRIPLE does not have a fixed semantics for object-oriented features such as classes and inheritance. TRIPLE's module system allows such features to be easily defined for different object-oriented and other data models like UML, Topic Maps, or RDF Schema. TRIPLE can be applied for instance to various e-learning tasks such as querying learning objects, inference for personalization, and ontology mapping.

**Features of Triple**

- Triple supports namespaces as well as the abbreviation of resources and namespaces. Thus the language can actually be used for the Semantic Web and TRIPLE rules become more readable and comprehensible.
- Triple supports so-called *Models*, i.e. sets of RDF statements which are true in a specific context. That means, in TRIPLE one can model that certain statements are hold only for specific context which allows very flexible modeling of knowledge and is essential for adding semantics to RDF in a layered approach by using rules. It is possible to apply set-theoretic operations to models, e.g. intersection, difference, and parameterize models.
- TRIPLE supports reification that means statements can be treated as objects themselves in rules.
- The rule language of TRIPLE is a syntactical extension of Horn-logic where literals are RDF statements (written in an F-Logic like syntax as molecules S[P->O]) and literals (i.e. RDF-statements) can be bound to models.
- In contrast to basic Horn-Logic, rules are allowed to have expressive bodies instead of simple conjunctions of literals. One can use a full First-order logic-style expressions for defining rule bodies.
- A rule without a body can be used to define facts. In this case, the rule arrow is omitted.
- Using Lloyd-Topor transformations one can transform TRIPLE rules to Horn logic rules. Thus every Horn logic engine can principally be used to evaluate TRIPLE rules.
- TRIPLE has both an ASCII and a RDF syntax.

An example shall illustrate TRIPLE and its capability to add semantic layers on top of RDF: An ontology *cars* can be specified in TRIPLE as a model car as follows:
Using a TRIPLE rule (without head), one can query the ontology for objects X which are declared as being subclasses of `xyz:MotorVehicle`:

```
FORALL X <- X[rdfs:subClassOf -> xyz:MotorVehicle]@cars.
```

What we can read of the result is that the RDF property `rdfs:subClassOf` in the ontology `cars` does not have a predefined semantics. In principle, it is just a syntactic construct. Thus, we do not get `xyz:MiniVan` as a result of the query.

How can we remedy this lack of semantics in our model?

TRIPLE supports the definition of semantic RDF extensions in an elegant modular way: For instance the semantics of RDF Schema and other (simple) frame systems can be directly defined in TRIPLE as a parameterized model. The semantics of OIL and DAML+OIL (i.e. expressive ontology languages) can be done by interaction with external reasoning components (e.g. DL classifiers) as well. In that way it is possible to layer semantics on top of RDF data sets.

A parameterized model `rdfschema(Mdl)` for defining the semantics of RDFS schema properties can roughly look like this, where the parameter `Mdl` represents the set of RDF statements to which the “semantic layer” should be added:
Using this model to enrich our ontology with the additional semantic layer, results in a different and intuitive result:

This reflects one of the main design goals for TRIPLE: it is possible to reason over heterogeneous RDF knowledge bases in one inference system. This particularly enables the integration of heterogeneous data sets that are represented in the simple RDF data model.

Providing reasoning systems for TRIPLE is not a difficult thing, because as mentioned above, TRIPLE rules can be mapped to Horn Logic and thus can be evaluated by any Horn Logic engine like XSB Prolog. An implementation can be downloaded at the TRIPLE Homepage.

References

http://www.dfki.uni-kl.de/frodo/triple
http://triple.semanticweb.org
3.3.2 OWL rule extensions

**OWL extension: SWRL**

*Overview*

The Semantic Web Rule Language (SWRL) is a proposal for a rule language for the Semantic Web based on a combination of the OWL DL and OWL Lite sublanguages of the OWL Web Ontology Language with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language. The proposal extends the set of OWL axioms to include Horn-like rules. It thus enables Horn-like rules to be combined with an OWL knowledge base. A high-level abstract syntax is provided that extends the OWL abstract syntax described in the OWL Semantics and Abstract Syntax document. An extension of the OWL model-theoretic semantics is also given to provide a formal meaning for OWL ontologies including rules written in this abstract syntax.

The proposed rules are of the form of an implication between an antecedent (body) and consequent (head). The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold.

Both the antecedent (body) and consequent (head) consist of zero or more atoms. An empty antecedent is treated as trivially true (i.e. satisfied by every interpretation), so the consequent must also be satisfied by every interpretation; an empty consequent is treated as trivially false (i.e., not satisfied by any interpretation), so the antecedent must also not be satisfied by any interpretation. Multiple atoms are treated as a conjunction. Note that rules with conjunctive consequents could easily be transformed via the Lloyd-Topor transformations [Lloyd, 1987] into multiple rules each with an atomic consequent.

Atoms in these rules can be of the form $C(x)$, $P(x,y)$, sameAs($x,y$) or differentFrom($x,y$), where $C$ is an OWL description, $P$ is an OWL property, and $x,y$ are either variables, OWL individuals or OWL data values. It is important to know that OWL DL becomes undecidable when extended in this way as rules can be used to simulate role value maps [Schmidt-Schauß, 1989].

An XML syntax is specified for these rules based on RuleML and the OWL XML presentation syntax. Furthermore, an RDF concrete syntax based on the OWL RDF/XML exchange syntax is defined.

*Abstract Syntax for Rules*

An OWL ontology in the abstract syntax contains a sequence of axioms and facts. Axioms may be of various kinds, e.g., subClass axioms and equivalentClass axioms. It is proposed to extend this with rule axioms.

```
axiom ::= rule
```

A rule axiom consists of an antecedent (body) and a consequent (head), each of which consists of a (possibly empty) set of atoms.

```
rule ::= "Implies( { annotation } antecedent consequent ')"
```
Informally, a rule may be read as meaning that if the antecedent holds (is "true"), then the consequent must also hold. An empty antecedent is treated as trivially holding (true), and an empty consequent is treated as trivially not holding (false). Rules with an empty antecedent can thus be used to provide unconditional facts; however such unconditional facts are better stated in OWL itself, i.e., without the use of the rule construct. Non-empty antecedents and consequents hold iff all of their constituent atoms hold, i.e., they are treated as conjunctions of their atoms. As mentioned above, rules with conjunctive consequents could easily transformed (via the Lloyd-Topor transformations) into multiple rules each with an atomic consequent.

\[
\text{atom} ::= \text{description}'(\ i-object\ ')' \\
| \text{individualvaluedPropertyID}'(\ i-object \ i-object\ ')' \\
| \text{datavaluedPropertyID}'(\ i-object \ d-object\ ')' \\
| \text{sameAs}'(\ i-object \ i-object\ ')' \\
| \text{differentFrom}'(\ i-object \ i-object\ ')
\]

Atoms can be of the form \(C(x), P(x,y), \text{sameAs}(x,y)\) or \(\text{differentFrom}(x,y)\), where \(C\) is an OWL description, \(P\) is an OWL property, and \(x, y\) are either variables, OWL individuals or OWL data values. In the context of OWL Lite, descriptions in atoms of the form \(C(x)\) may be restricted to class names.

Informally, an atom \(C(x)\) holds if \(x\) is an instance of the class description \(C\), an atom \(P(x,y)\) holds if \(x\) is related to \(y\) by property \(P\), an atom \(\text{sameAs}(x,y)\) holds if \(x\) is interpreted as the same object as \(y\), and an atom \(\text{differentFrom}(x,y)\) holds if \(x\) and \(y\) are interpreted as different objects. Note that the latter two forms can be seen as "syntactic sugar": they are convenient, but do not increase the expressive power of the language (i.e., such (in)equalities can already be expressed using the combined power of OWL and rules without explicit (in)equality atoms).

\[
\text{i-object} ::= \text{i-variable} \mid \text{individualID} \\
\text{d-object} ::= \text{d-variable} \mid \text{dataLiteral}
\]

Atoms may refer to individuals, data literals, individual variables or data variables. Variables are treated as universally quantified, with their scope limited to a given rule. As usual, only variables that occur in the antecedent of a rule may occur in the consequent (a condition usually referred to as "safety").

\[
\text{i-variable} ::= I-variable(' URIreference ')
\]
\[
\text{d-variable} ::= D-variable(' URIreference ')
\]

While the abstract EBNF syntax is consistent with the OWL specification, and is useful for defining XML and RDF serialisations, it is rather verbose and not particularly easy to read. In the following we will, therefore, often use a relatively informal "human readable" form similar to that used in many published works on rules.

In this syntax, a rule has the form: \[\text{antecedent} \Rightarrow \text{consequent}\]
where both \(\text{antecedent}\) and \(\text{consequent}\) are conjunctions of atoms written \(a_1 \wedge \ldots \wedge a_n\). Variables are indicated using the standard convention of prefixing them with a question mark (e.g., '?x'). Using this syntax, a rule asserting that the composition of parent and brother properties implies the uncle property would be written:
parent(?x,?y) ∧ brother(?y,?z) ⇒ uncle(?x,?z)

Examples

(1) A simple use of these rules would be to assert that the combination of the \texttt{hasParent} and \texttt{hasBrother} properties implies the \texttt{hasUncle} property. Informally, this rule could be written as:

\[
\text{hasParent}(\textit{x1}, \textit{x2}) \land \text{hasBrother}(\textit{x2}, \textit{x3}) \Rightarrow \text{hasUncle}(\textit{x1}, \textit{x3})
\]

In the abstract syntax the rule would be written like:

\[
\text{Implies}(\text{Antecedent}(\text{hasParent}(\text{I-variable(x1)} \text{I-variable(x2)}) \\
\text{hasBrother}(\text{I-variable(x2)} \text{I-variable(x3)))) \\
\text{Consequent}(\text{hasUncle}(\text{I-variable(x1)} \text{I-variable(x3))))})
\]

From this rule, if John has Mary as a parent and Mary has Bill as a brother then John has Bill as an uncle.

(2) A very common use for rules is to move property values from one individual to a related individual, as in the following example that expresses the fact that the style of an art object is the same as the style of the creator.

\[
\text{Artist}(\textit{x}) \land \text{artistStyle}(\textit{x}, \textit{y}) \land \text{Style}(\textit{y}) \land \text{creator}(\textit{z}, \textit{x}) \Rightarrow \\
\text{style/period}(\textit{z}, \textit{y})
\]

Or in abstract syntax:

\[
\text{Implies}(\text{Antecedent}(\text{Artist}(\text{I-variable(x)})) \\
\text{artistStyle}(\text{I-variable(x)} \text{I-variable(y)}) \\
\text{Style}(\text{I-variable(y)}) \\
\text{creator}(\text{I-variable(z)} \text{I-variable(x)})) \\
\text{Consequent}(\text{style/period}(\text{I-variable(z)} \text{I-variable(y)})))
\]

The XML and RDF concrete syntax is specified in \url{http://www.daml.org/2003/11/swrl/}.

Summary

SWRL represents a straightforward extension of OWL-DL by Horn-rules. More precisely, the allowed Horn-rules fall into the Datalog sublanguage of Horn-Logic. This increases the expressivity of the OWL language and it’s modeling capabilities significantly. However, checking subsumption in the extended language is undecidable in general, that means that one looses computational garuantees.

We are not aware of any implementation of a system supporting

Resources

\url{http://www.daml.org/2003/11/swrl/} \\
\url{http://www.w3.org/TR/owl-features/} \\
\url{http://www.ruleml.org/}
3.4 Candidate solution

In the previous sections, the characteristics of the different languages have been presented making a special effort to show the advantages and disadvantages of the different candidates for the Esperonto project.

Given the nature of the project and the position (evolution/characteristics/impact) of the languages, the most convenient solution would still be to use some of the OWL languages. The reasons that support this decision are the following:

- **Scope and Visibility.** OWL is meant to be “the” Semantic Web language given the amount of efforts involved in its development, and the characteristics presented by the language so far.

- **Platform independence.** The Esperonto ontology server which is based on WebODE platform is language independent, providing translators and APIs for ontology access. To give more impact within the community to the project OWL is a good choice.

- **Flexibility.** When needed OWL could be translated into a more simple language, e.g. RDF, however, this implies a loss of information, so it is not recommended, but in case the communications facilities/protocols require it could be achieved, since OWL is in top layers of the semantic cake.

- **Alignment.** Even though ontology aligning should be language independent OWL already counts with built in primitives that could easy the process.

- **Reasoning support.** It could happen that reasoners do not become available by the time they are needed. If this is the case the consortium agreed in developing such tools.

A smart combination of the different OWL languages should be the ideal for the project. Taking into account the development state of the different sub-languages, and the requirements of the Esperonto project, RDF(S) expressive and inference power was enough for the first prototype. In following versions and given the flexibility of the Ontology server to incorporate one language or another, it can be reconsidered, thus all types of OWL-languages can be used.

Nonetheless, the WSMO initiative is working on some new ontology languages, particularly the OWL- family of languages, which are interesting for Esperonto in the near future, since they have interesting computational properties. Most notably, it is expected that these languages will be backed by reasoning tools which work efficient even for ontologies with large sets of instances. In particular, it would be interesting for example to explore how for instance the Local-Closed-World Assumption of OWL-Flight could be applied and exploited within the Esperonto use cases. Unfortunately, the initiative is still in its early stage and thus the languages are partly not fully defined yet and lack of available reasoning tools. The final goal of the WSMO initiative is to come up with a standard proposal for a Semantic Description Framework for Web Services as well as a reference implementation. Therefore, a standardization for the ontology languages developed in the context of WSMO and WSML can be expected as well.
4. Conclusions and Future Plan

The Web ontology language must be able to describe and organize knowledge in the Web in the machine understandable way. It is a very complicated task, which can be simplified by defining categories via creating concepts, frames, classes, rules, etc. However, taken to the extreme, organizing knowledge requires the facilities of the logical formalism, which can deal with temporal, spatial, epistemic, and inferential aspects of knowledge. Implementations of the Web ontology language must provide these inference services, making them much more than just simple data storage and retrieval systems. In certain application-dependent cases, Web ontology language must have a clear model theory to secure the precise determination meanwhile the semantics of the OWL-family languages involves generalization of data models, like the relational data model or semi-structured data, which can deal with identifiable objects and uncertain and vague information. Tools are needed for the massive adoption of the languages themselves, therefore the easy-of-use for the programmers to implement all the essential features of the web ontology languages into the supporting tools are very important as well.

Developing real semantics in the scope of the Semantic Web language tower as sketched by Tim Berners-Lee requires much more work. Currently many layering ideas oriented to syntactical and semantic extensions compete with each other. Choosing and clarifying the differences among the different layers becomes a difficult task comparable to developing one more layer for the language.

In this deliverable the main improvements in the Semantic Web Languages area have been sketched, with the clear aim of helping and justification the decision making process of which language to use in the Esperonto project. At the moment the small quantity of tools and examples in OWL and other new promising languages makes choosing them a tough decision. Nevertheless, it is clear that at the moment OWL language is the most promising for the project. Clearly, there are interesting new languages that are in the process of being invented and designed at the moment, but all these languages are in a very early and immature state for actual applications. Most of them lack in particular reasoning support. In order to achieve the maximum degree of visibility for our work; the work carried out in the project should be based on the OWL language. Nonetheless, we should carefully follow the development of the new languages discussed in Section 2. Perhaps, a smart combination of various languages could also be a possibility in our case depending of the concrete problem to be solved.

In the next version of this deliverable, we will keep the sharp eye on the newly development and evolving in the Semantic Web ontology language area and document and discuss the pros and cons to provide a support to the future versions of tools of the Esperonto project origin and similar.
5. References


