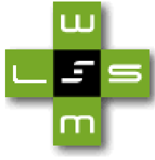

D16.1v0.2 The Web Service Modeling Language WSML

WSML Working Draft 16 March 2005

This version
http://www.wsmo.org/TR/d16/d16.1/v0.2/20050316/
Latest version
http://www.wsmo.org/TR/d16/d16.1/v0.2/
Previous version
http://www.wsmo.org/TR/d16/d16.1/v0.2/20050314/

Editor:
Jos de Bruijn
Authors:
Jos de Bruijn
Holger Lausen
Reto Krummenacher
Axel Polleres
Livia Predoiu
Michael Kifer
Dieter Fensel
Reviewers:
Ian Horrocks
Jeff Pan

Copyright © 2005 DERI ®, All Rights Reserved. DERI liability, trademark, document use, and software licensing rules apply.

Abstract

We introduce the Web Service Modeling Language WSML which provides a formal syntax and semantics for the Web Service Modeling Ontology WSMO. WSML is based on different logical formalisms, namely, Description Logics, First-Order Logic and Logic Programming, which are useful for the modeling of Semantic Web services.

WSML consists of a number of variants based on these different logical formalisms, namely WSML-Core, WSML-DL, WSML-Flight, WSML-Rule and WSML-Full.

WSML-Core corresponds with the intersection of Description Logic and Horn Logic (without function symbols and without equality), extended with datatype support in order to be useful in practical applications. WSML-Core is fully compliant with a subset of OWL.

WSML-Core is extended, both in the direction of Description Logics and in the direction of Logic Programming, to WSML-DL and WSML-Flight.

WSML-DL extends WSML-Core to an expressive Description Logic, namely, *SHIQ*, thereby covering that part of OWL which is efficiently implementable.

WSML-Flight extends WSML-Core in the direction of Logic Programming. WSML-Flight has a rich set of modeling primitives for modeling different aspects of attributes, such as value constraints and integrity constraints. Furthermore, WSML-Flight incorporates a fully-fledged rule language, while still allowing efficient decidable reasoning. To be more precise, WSML-Flight allows to write down any Datalog rule, extended with inequality and (locally) stratified negation.

WSML-Rule extends WSML-Flight to a fully-fledged Logic Programming language, including function symbols. WSML-Rule no longer restricts the use of variables in logical expressions.

The final WSML variant unifies the Description Logic and Logic Programming paradigms.

WSML-Full unifies all WSML variants under a common First-Order umbrella with non-monotonic extensions which allow to capture nonmonotonic negation of WSML-Rule.

All WSML variants are described in terms of a normative human-readable syntax. Besides the human-readable syntax we provide an XML and an RDF syntax for exchange between machines. Furthermore, we provide a mapping between WSML ontologies and OWL for basic inter-operation with OWL ontologies through a common semantic subset of OWL and WSML.

This deliverable supersedes a number of now obsolete WSML deliverables. References to these now obsolete deliverables can be found at: http://www.wsmo.org/TR/d16/

Table of Contents

PART I: PRELUDE

1. Introduction

- 1.1. Structure of the deliverable

PART II: WSML VARIANTS

2 WSML Syntax

- 2.1 WSML Syntax Basics
  - 2.1.1 Namespaces in WSML
  - 2.1.2 Identifiers in WSML
  - 2.1.3 Comments in WSML
- 2.2 WSML Elements
  - 2.2.1 WSML Variant Declaration
  - 2.2.2 Namespace References
  - 2.2.3 Header
- 2.3 Ontology Specification in WSML
  - 2.3.1 Concepts
  - 2.3.2 Relations
  - 2.3.3 Instances
  - 2.3.4 Axioms
- 2.4 Capability and Interface Specification in WSML
  - 2.4.1 Capabilities
  - 2.4.2 Interfaces
- 2.5 Goal Specification in WSML
- 2.6 Mediator Specification in WSML
  - 2.6.1 ooMediators
  - 2.6.2 wwMediators
  - 2.6.3 ggMediators
  - 2.6.4 wgMediators
- 2.7 Web Service Specification in WSML
- 2.8 General Logical Expressions in WSML

3 WSML-Core

- 3.1 Basic WSML-Core Syntax
- 3.2. WSML-Core Ontologies
  - 3.2.1 Concepts
  - 3.2.2 Relations
  - 3.2.3 Instances
  - 3.2.4 Axioms
- 3.3. Goals in WSML-Core
- 3.4. Mediators in WSML-Core
- 3.5. Web Services in WSML-Core
- 3.6. WSML-Core Logical Expression Syntax

4 WSML-DL

- 4.1 Basic WSML-DL Syntax
- 4.2. WSML-DL Ontologies
- 4.3. Goals in WSML-DL
- 4.4. Mediators in WSML-DL
- 4.5. Web Services in WSML-DL
- 4.6. WSML-DL Logical Expression Syntax
- 4.7. Differences between WSML-Core and WSML-DL

5 WSML-Flight

- 5.1 Basic WSML-Flight Syntax
- 5.2. WSML-Flight Ontologies
- 5.3. Goals in WSML-Flight
- 5.4. Mediators in WSML-Flight
- 5.5. Web Services in WSML-Flight
- 5.6. WSML-Flight Logical Expression Syntax
- 5.7. Differences between WSML-Core and WSML-Flight

6 WSML-Rule

- 6.1. WSML-Rule Logical Expression Syntax
- 6.2. Differences between WSML-Flight and WSML-Rule

7 WSML-Full

- 7.1. Differences between WSML-DL and WSML-Full
- 7.2. Differences between WSML-Rule and WSML-Full

8. WSML Semantics

- 8.1. Mapping Conceptual Syntax to Logical Expression Syntax
- 8.2. Preprocessing steps
- 8.3. WSML-Core Semantics
- 8.4. WSML-Flight Semantics
- 8.5. WSML-Rule Semantics

PART III: THE WSML EXCHANGE SYNTAXES

9 XML Syntax for WSML

10 RDF Syntax for WSML

11 Mapping to OWL

- 11.1. Mapping WSML-Core to OWL DL
- 11.2. Mapping OWL DL to WSML-Core

PART IV: FINALE

12. Related Efforts

Appendix A. Human-Readable Syntax

- A.1. BNF-Style Grammar
- A.2. Example of Human-Readable Syntax

Appendix B. Schemas for the XML Exchange Syntax

Appendix C. Built-ins in WSML

- C.1. WSML Datatypes
- C.2. WSML Datatype Predicates
- C.3. Translating Built-in Symbols to Datatype Predicates

Appendix D. WSML Keywords

Appendix E. Relation to WSMO Conceptual Model

Appendix F. Changelog

References

Acknowledgements

PART I: PRELUDE

1. Introduction

The Web Service Modeling Ontology WSMO [Roman *et al.* 2004] proposes a conceptual model for the description of Ontologies, Semantic Web services, Goals, and Mediators, providing the conceptual grounding for Ontology and Web service descriptions. In this document we take the conceptual model of WSMO as a starting point for the specification of a family of Web Service description and Ontology specification languages. The Web Service Modeling Language (WSML) aims at providing means to formally describe all the elements defined in WSMO. The different variants of WSML correspond with different levels of logical expressiveness and the use of different languages paradigms. More specifically, we take Description Logics, First-Order Logic and Logic Programming as starting points for the development of the WSML language variants. The WSML language variants are both syntactically and semantically layered. All WSML variants are specified in terms of a human-readable syntax with keywords similar to the elements of the WSMO conceptual model. Furthermore, we provide XML and RDF exchange syntaxes, as well as a mapping between WSML ontologies and OWL ontologies for interoperability with OWL-based applications.

Ontologies and Semantic Web services need formal languages for their specification in order to enable automated processing. As for ontology descriptions, the W3C recommendation for an ontology language OWL [Dean & Schreiber, 2004] has limitations both on a conceptual level and with respect to some of its formal properties [de Bruijn et al., 2005]. One proposal for the description of Semantic Web services is OWL-S [OWL-S, 2004]. However, it turns out that OWL-S has serious limitations on a conceptual level and also, the formal properties of the language are not entirely clear [Lara et al., 2005]. For example, OWL-S offers the choice between different languages for the specification of preconditions and effects. However, it is not entirely clear how these languages interact with OWL, which is used for the specification of inputs and output. These unresolved issues were the main motivation to provide an alternative, unified language for WSMO.

1.1. Structure of the Document

This document is further structured as follows:

PART II: WSML VARIANTS

Chapter 2 describes the general WSML modeling elements, as well as syntax basics, such as the use of namespaces and the basic vocabulary of the languages. Further chapters define the restrictions imposed by the different WSML variants on this general syntax. Chapter 3 describes WSML-Core, which is the least expressive of the WSML variants. WSML-Core is based on the intersection of Description Logics and Logic Programming and can thus function as the basic interoperability layer between both paradigms. Chapter 4 presents WSML-DL, which is an extension of WSML-Core. WSML-DL is a full-blown Description Logic and offers similar expressive power to OWL DL [Patel-Schneider et al., 2004]. Chapter 5 describes WSML-Flight, which is an extension of WSML-Core in the direction of Logic Programming. WSML-Flight is a more powerful language and offers more expressive modeling constructs than WSML-Core. The extension described by WSML-Flight is disjoint from the extension described by WSML-DL. Chapter 6 describes WSML-Rule,which is a full-blown Logic Programming language; WSML-Rule allows the use of function symbols and does not require rule safety. It is an extension of WSML-Flight and thus it offers the same kind of conceptual modeling features. Chapter 7, finally, presents WSML-Full which is a superset of both WSML-Rule and WSML-DL. WSML-Full can be seen as a notational variant of First-Order Logic with nonmonotonic extensions. Finally, Chapter 8 defines the WSML semantics.

PART III: THE WSML EXCHANGE SYNTAXES

The WSML variants are described in terms of their normative human-readable language in PART II. Although this syntax has been formally specified in the form of a grammar (see also Appendix A), there are limitations with respect to the exchange of the syntax over the Web. Therefore, Chapter 9 presents the XML exchange syntax for WSML, which is the preferred syntax for the exchange of WSML specifications between machines. Chapter 10 describes the RDF syntax of WSML, which can be used by RDF-based applications. Chapter 11, finally, describes a mapping between WSML and OWL ontologies in order to allow interoperation with OWL-based applications.

PART IV: FINALE

We conclude the document with describing efforts related to the WSML language in Chapter 12. These related efforts are mostly concerned with implementation of WSML-based tools and tools utilizing WSML for specific purposes.

Appendix Guide

This document contains a number of appendices:

Appendix A consists of the formal grammar shared by all WSML variants, as well as a complete integrated example WSML specification to which references are made in the various chapters of this document. Appendix B contains references to the XML Schemas, the XML Schema documentation and the XML version of the WSML example from Appendix A. These documents are all online resources. Appendix C describes the built-in predicates and datatypes of WSML, as well as a set of infix operators which correspond with particular built-in predicates. Appendix D contains a complete list of WSML keywords, as well as references to the sections in the document where these are described. Appendix E describes the relation between WSML and the WSMO conceptual model. Finally, Appendix F contains the changelog which documents the changes between the current and previous version of this document.

PART II: WSML VARIANTS

Figure 1 shows the different variants of WSML and the relationship between the variants. In the figure, an arrow stands for "extension in the direction of". The variants differ in the logical expressiveness they offer and in the underlying language paradigm. 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.

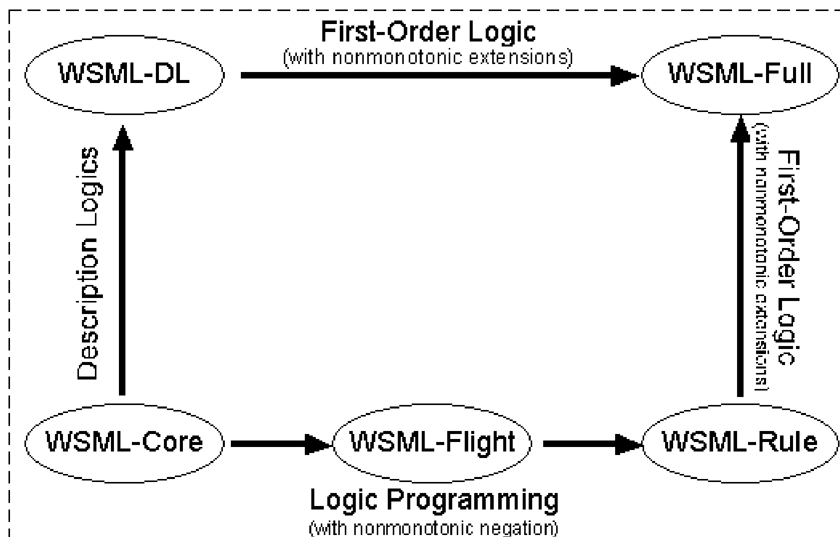

Figure 1. WSML Space

WSML-Core
: This language is defined by the intersection of Description Logic and Horn Logic, based on Description Logic Programs [Grosof et al., 2003]. It has the least expressive power of all the languages of the WSML family and therefore has 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 support for datatypes and datatype predicates.[2] WSML-Core is based on the intersection of Description Logics and Datalog, corresponding to the DLP fragment [Grosof et al., 2003].

WSML-DL
: This language is an extension of WSML-Core which fully captures the Description Logic *SHIQ*(D), which captures a major part of the (DL species of the) Web Ontology Language OWL [Dean & Schreiber, 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 such features as meta-modeling, constraints and nonmonotonic negation. WSML-Flight is based on a logic programming variant of F-Logic [Kifer et al., 1995] and is semantically equivalent to Datalog with inequality and (locally) stratified negation. As such, WSML-Flight provides a powerful rule language.

WSML-Rule
: This language is an extension of WSML-Flight in the direction of Logic Programming. The language captures several extensions such as the use of function symbols and unsafe rules.

WSML-Full
: WSML-Full unifies WSML-DL and WSML-Rule under a First-Order umbrella with extensions to support the nonmonotonic negation of WSML-Rule. It is yet to be investigated which kind of formalisms are required to achieve this. Possible formalisms are Default Logic, Circumscription and Autoepistemic Logic.

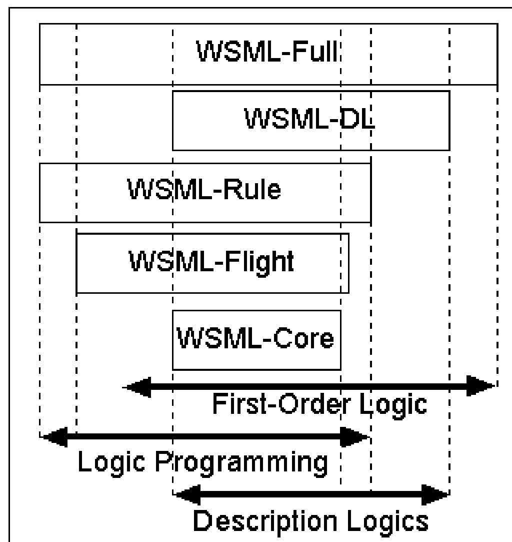

Figure 2. WSML Layering

As can be seen from Figure 2, WSML has two alternative layerings, namely, WSML-Core -> WSML-DL -> WSML-Full and WSML-Core -> WSML-Flight -> WSML-Rule -> 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 the 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 in this document are WSML-Core, WSML-Flight and WSML-Rule. WSML-DL will correspond (semantically) with the Description Logic *SHIQ*(D_n), extended with more extensive datatype support.

In the descriptions in the subsequent chapters we use fragments of the WSML grammar (see Appendix A.1 for the full grammar) in order to show the syntax of the WSML elements. The grammar is specified using a dialect of Extended BNF which can be used directly in the SableCC compiler compiler [SableCC]. Terminals are delimited with single quotes, non-terminals are underlined and refer to the corresponding productions. Alternatives are separated using vertical bars '|'; optional elements are appended with a question mark '?'; elements that may occur zero or more times are appended with an asterisk '*'; elements that may occur one or more times are appended with a plus '+'. In the case of multiple references to the same non-terminal in a production, the non-terminals are disambiguated by using labels of the form '[*label*]:'. In order to keep the descriptions in this Part concise, we do not fully describe all non-terminal. Non-terminals are linked to the grammar in Appendix A.

Throughout the WSML examples in the following chapters, we use boldfaced text to distinguish WSML keywords.

2 WSML Syntax

In this chapter we introduce the WSML syntax. The general WSML syntax captures all features of all WSML variants and thus corresponds with the syntax for WSML-Full. Subsequent chapters will define restrictions on this syntax for the specification of specific WSML variants.

The WSML syntax consists of two major parts: the conceptual syntax and the logical expression syntax. The conceptual syntax is used for the modeling of ontologies, goals, web services and mediators; these are the elements of the WSMO conceptual model. Logical expressions are used to refine these definitions using a logical language.

A WSML document has the following structure:

```
wsml       =  wsmlvariant? namespace? definition*
definition =   goal
             | ontology
             | webservice
             | mediator
```

This chapter is structured as follows. The WSML syntax basics, such as the use of namespaces, identifiers, etc., are described in Section 2.1. The elements in common between all WSML specifications are described in Section 2.2. WSML ontologies are described in Section 2.3. The elements shared between goals and web services, namely, capabilities and interfaces, are described in Section 2.4. Goals, mediators and web services are described in Sections 2.5, 2.6 and 2.7, respectively. Finally, the WSML logical expression syntax is specified in Section 2.8.

2.1 WSML Syntax Basics

The conceptual 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.

It is often possible to specify a list of arguments, for example for attributes. Such argument lists in WSML are separated by commas and surrounded by curly brackets. Statements in WSML 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, non-functional properties (annotations), import of ontologies, references to used mediators and the type of the specification. This meta-information block is strictly ordered. The second part 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 goal or web service specification), etc., is not ordered.

The remainder of this section explains the use of namespaces, identifiers and datatypes in WSML. Subsequent sections explain the different kinds of WSML specifications and the WSML logical expression syntax.

2.1.1 Namespaces in WSML

Namespaces were first introduced in XML [XML-NAMESPACES-1.1] for the purpose of qualifying names which originate from different XML documents. In XML, each qualified name consists of a tuple <namespace, localname>. RDF adopts the mechanism of namespaces from XML with the difference that qualified names are not treated as tuples, but rather as abbreviations for full URIs.

WSML adopts the namespace mechanism of RDF. A namespace can be seen as part of an IRI (see the next Section). The WSML keywords themselves fall in the namespace http://www.wsmo.org/wsml/wsml-syntax# (commonly abbreviated as 'wsml').

Namespaces can be used to syntactically distinguish elements of multiple WSML specifications and, more generally, resources on the Web. A namespace denotes a syntactical domain for naming resources.

Whenever a WSML specification has a specific identifier which corresponds to a Web address, it is good practice to have a relevant document on the location pointed to by the identifier. 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 by the identifier.

2.1.2 Identifiers in WSML

An identifier in WSML is either a data value, an IRI [Duerst & Suignard, 2005], or an anonymous ID.

Data values

WSML has direct support for different types of concrete data, namely, strings, integers and decimals, which correspond to the XML Schema [Biron & Malhorta, 2004] primitive datatypes *string*, *integer* and *decimal*. These concrete values can then be used to construct more complex datatypes, corresponding to other XML Schema primitive and derived datatypes, using datatype constructor functions. See also Appendix C.

WSML uses datatype wrappers to construct data values based on XML Schema datatypes. The use of datatype wrappers gives more control over the structure of the data values than the lexical representation of XML. For example, the date: 3rd of February, 2005, which can be written in XML as: '2005-02-03', is written in WSML as: _date(2005,2,3). The arguments of such a term can be either strings, decimals, integers or variables. No other arguments are allowed for such data terms. Each conforming WSML implementation is required to support at least the *string*, *integer* and *decimal* datatypes.

Datatype identifiers manifest themselves in WSML in two distinct ways, namely, as concept identifers and as datatype wrappers. A datatype wrapper is used as a datastructure for capturing the different components of data values. Datatype identifiers can also be used directly as concept identifiers. Note however that the domain of interpretation of any datatype is finite and that asserting membership of a datatype for a value which does not in fact belong to this datatype, leads to an inconsistency in the knowledge base.

An number of example data values:

```
_date(2005,3,12)
_sqname("http://www.wsml.org/wsml/wsml-syntax#", "goal")
_boolean("true")
```

The following are example attribute definitions which restrict the range of the attribute to a particular datatype:

```
age ofType _integer
location ofType _iri
hasChildren ofType _boolean
```

WSML allows the following syntactical shortcuts for particular datatypes:

- Data values of type *string* can be written between double quotes "". Double quotes inside a string should be escaped using the '\' ("backslash"): \".
- Integer values can simply be written as such. Thus an integer of the form *integer* is a shortcut for _integer("*integer*"). For example, 4 is a shortcut for _integer("*4*")
- Decimal values can simply be written as such, using the '.' as decimal symbols. Thus a literal of the form *decimal* is a shortcut for _decimal("*decimal*"). For example, 4.2 is a shortcut for _decimal("*4.2*").

```
integer        =  '-'?  digit+
decimal        =  '-'?  digit+  '.'  digit+
string         = '"'  string_content*  '"'
string_content = escaped_char | not_escape_char_not_dquote
```

Appendix C lists the built-in predicates which are required to be supported by any conforming WSML application, as well as the infix notation which serves as a shortcut for the built-ins.

Furthermore, WSML also allows shortcut syntax for IRI and sQName data values, as described below.

Internationalized Resource Identifiers

The *IRI* (Internationalized Resource Identifier) [Duerst & Suignard, 2005] 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 IRIs starting with 'urn:isbn:'). The IRI proposed standard is a successor to the popular URI standard. In fact, every URI is an IRI.

An IRI can be abbreviated to an sQName. Note that the term 'QName' has been used, after its introduction in XML [Bray et al., 2004], with different meanings. The meaning of the term 'QName' as defined in XML especially got blurred after the adoption of the term in RDF. In XML, QNames are simply used to qualify local names and thus every name is a tuple <namespace, localname>. In RDF, QNames have become abbreviations for URIs, which is different from the meaning in XML. WSML adopts a view similar to the RDF-like version of QNames, but due to its deviation from the original definition in XML we call them sQNames which is short for 'serialized QName'.

An sQName is equivalent to the IRI which is obtained by concatenating the namespace IRI (to which the prefix refers) with the local part of the sQName. Therefore, an sQName can be seen as an abbreviation for a IRI which enhances the legibility of the specification. In case an sQName has no prefix, the namespace of the sQName is the default namespace of the document.

An sQName consists of two parts, namely, the namespace prefix and the local part. WSML allows two distinct ways to write sQNames. sQName can be seen as a datatype and thus it has an associated datatype wrapper, namely, _sqname (see also Appendix C), which has two arguments: namespace and localname. Because sQNames are very common in WSML specifications, WSML allows a short syntax for sQNames. An sQName can simply be written using a namespace prefix and a localname, separated by a hash ('#'): *namespace_prefix#localname*. It is also possible to omit the namespace prefix and the hash symbol. In this case, the name is defined in the default namespace.

IRI is a datatype in WSML and has the associated datatype wrapper _iri with one argument (the IRI). For convenience, WSML also allows a short form with the delimiters ' _" ' and ' " '. For convenience, a sQName does not require special delimiters. However, sQNames may not coincide with any WSML keywords. The characters '.' and '-' in an sQName need to be escaped using a '\':

```
full_iri  =  '_"' iri_reference '"'
sQName    =  (name '#')? name
iri       =  full_iri
           | sqname
```

Examples of full IRIs in WSML:

```
_"http://example.org/PersonOntology#Human"
_"http://www.uibk.ac.at/"
```

Examples of sQNames in WSML (with corresponding full IRIs; dc stands for http://purl.org/dc/elements/1.1#, foaf stands for http://xmlns.com/foaf/0.1/ and xsd stands for http://www.w3.org/2001/XMLSchema#; we assume the default namespace http://example.org/#):

- dc#title (http://purl.org/dc/elements/1.1#title)
- foaf#name (http://xmlns.com/foaf/0.1/name)
- xsd#string (http://www.w3.org/2001/XMLSchema#string)
- Person (http://example.org/#Person)
- hasChild (http://example.org/#hasChild)

WSML defines the following two IRIs: http://www.wsmo.org/wsml/wsml-syntax#true and http://www.wsmo.org/wsml/wsml-syntax#false, which stand for universal truth and universal falsehood, respectively. Note that for convenience we typically use the abbreviated sQName form (where wsml stands for the default WSML namespace http://www.wsmo.org/wsml/wsml-syntax#): wsml#true, wsml#false. Additionally, WSML allows the keywords 'true' and 'false' in the human-readable syntax.

Please note that the IRI of a resource does not necessarily correspond to a document 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.

Anonymous identifiers

An anonymous identifier represents a IRI which is meant to be globally unique. Global uniqueness is to be ensured by the system interpreting the WSML description (instead of the author of the specification). It can be used whenever the concrete identifier to be used to denote an object is not relevant, but when we require the identifier to be new (i.e., not used anywhere else in the WSML description).

Anonymous identifiers in WSML follow the naming convention for anonymous IDs presented in [Yang & Kifer, 2003]. Unnumbered anonymous IDs are denoted with '_#'. Each occurrence of '_#' denotes a new anonymous ID and different occurrences of '_#' are unrelated. Thus each occurrence of an unnumbered anonymous ID can be seen as a new unique identifier.

Numbered anonymous IDs are denoted with '_#*n*' where *n* stands for an integer denoting the number of the anonymous ID. The use of numbered anonymous IDs is limited to logical expressions and can therefore not be used to denote entities in the conceptual syntax. Multiple occurrences of the same numbered anonymous ID within the same logical expression are interpreted as denoting the same object.

```
anonymous    = '_#'
nb_anonymous = '_#' digit+
```

Take as an example the following logical expressions:

```
_#[a hasValue _#1] and _#1 memberOf b.

_#1[a hasValue _#] and _# memberOf _#.
```

There are in total three occurrences of the unnumbered anonymous ID. All occurrences are unrelated. Thus, the second logical expression *does not* state that an object is a member of itself, but rather that some anonymous object is a member of some other anonymous object. The two occurrences of _#1 in the first logical expression denote the same object. Thus the value of attribute a is a member of b. The occurrence of _#1 in the second logical expression is, however, not related to the occurrence of _#1 in the first logical expression.

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 was found.
- In case the identifier of a WSML element (e.g., concept, relation, postcondition) has been omitted, the unnumbered anonymous identifier '_#' is used to identify the element.

```
id     = iri
       | anonymous
       | 'true'
       | 'false'
idlist = id
       | '{' id ( ',' id )* '}'
```

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.

Note that the sets of identifiers for the top-level elements, namely *ontology*, *goal*, *webService*, *ooMediator*, *ggMediator*, *wgMediator* and *wwMediator*, are pairwise disjoint and also disjoint from all other identifiers.

2.1.3 Comments in WSML

A WSML file may at any place contain a comment. A single line comment starts with **comment** or // and ends with a line brake. Comments can also range over multiple lines, in which they need to be delimited by /* and */.

```
comment       = short_comment | long_comment
short_comment = ('//' | 'comment ') not_cr_lf* eol
long_comment  = '/*' long_comment_content* '*/'
```

It is recommended to use non-functional properties for any information related to the actual WSML descriptions; comments should be only used for meta-data about the WSML file itself. Comments are disregarded when parsing the WSML document.

An number of examples:

```
/* Illustrating a multi line
 * comment
 */

// a one-line comment
comment another one-line comment
```

2.2 WSML Elements

This section describes the elements in common between all types of 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.

2.2.1 WSML Variant

Every WSML specification document may start with the **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
WSML-Core	http://www.wsmo.org/wsml/wsml-syntax/wsml-core
WSML-Flight	http://www.wsmo.org/wsml/wsml-syntax/wsml-flight
WSML-Rule	http://www.wsmo.org/wsml/wsml-syntax/wsml-rule
WSML-DL	http://www.wsmo.org/wsml/wsml-syntax/wsml-dl
WSML-Full	http://www.wsmo.org/wsml/wsml-syntax/wsml-full

Table 2.1: WSML variant identifiers

The specification of the **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.

wsmlvariant =	'wsmlVariant'	full_iri

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

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/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.

2.2.2 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 followed 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 '{' '}'. In case only a default namespace is declared, the curly brackets are not required.

namespace	=	'namespace' prefixdefinitionlist
prefixdefinitionlist	=	full_iri
	|	'{' prefixdefinition (',' prefixdefinition)* '}'
prefixdefinition	=	name full_iri
	|	full_iri

Two examples, one with a number of namespace declarations and one with only a default namespace:

```
namespace {_"http://www.example.org/ontologies/example#",
     dc _"http://purl.org/dc/elements/1.1#",
     foaf _"http://xmlns.com/foaf/0.1/",
     xsd _"http://www.w3.org/2001/XMLSchema#",
     wsml _"http://www.wsmo.org/wsml-syntax#",
     loc _"http://www.wsmo.org/ontologies/location#",
     oo _"http://example.org/ooMediator#"}

namespace _"http://www.example.org/ontologies/example#"
```

2.2.3 Header

Any WSML specification may have non-functional properties, may import ontologies and may use mediators:

```
header =  nfp
        | importsontology
        | usesmediator
```

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 delimited with the keywords **nonFunctionalProperties** and **endNonFunctionalProperties** or the short forms **nfp** and **endnfp**. 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 data value, an anonymous identifier or a comma-separated list of the former, delimited with curly brackets. 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/wsml/wsml-syntax#). These properties are: **wsml#version**, **wsml#accuracy**, **wsml#financial**, **wsml#networkRelatedQoS**, **wsml#performance**, **wsml#reliability**, **wsml#robustness**, **wsml#scalability**, **wsml#security**, **wsml#transactional**, **wsml#trust** (here we assume that the prefix wsml has been defined as referring to the WSML namespace; see Section 2.1.1). For recommended usage of these properties see [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.

```
nfp =  'nfp'  attributevalue*  'endnfp'
     | 'nonFunctionalProperties'  attributevalue*  'endNonFunctionalProperties'
```

An example:

```
nonFunctionalProperties
  dc#title hasValue "WSML example ontology"
  dc#subject hasValue "family"
  dc#description hasValue "fragments of a family ontology to provide WSML examples"
  dc#contributor hasValue { _"http://homepage.uibk.ac.at/~c703240/foaf.rdf",
     _"http://homepage.uibk.ac.at/~csaa5569/",
     _"http://homepage.uibk.ac.at/~c703239/foaf.rdf",
     _"http://homepage.uibk.ac.at/homepage/~c703319/foaf.rdf" }
  dc#date hasValue _date("2004-11-22")
  dc#format hasValue "text/html"
  dc#language hasValue "en-US"
  dc#rights hasValue _"http://www.deri.org/privacy.html"
  wsml#version hasValue "$Revision: 1.156 $"
endNonFunctionalProperties
```

Non-functional properties in WSML are not part of the logical language; programmatic access to these properties can be provided through an API.

Importing Ontologies

Ontologies may be imported in any WSML specification through the import 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** annotation property in OWL. This means the resulting ontology is the union of all axioms and definitions in the importing and imported ontologies. 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:

```
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.

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 Section 2.5. We are here only concerned with how mediators can be referenced from a WSML specification. Mediators are currently underspecified and thus this reference to the use of mediators can be seen as a placeholder.

The (optional) used mediators block is identified by the keywords **usesMediator** which is followed by one or more identifiers of WSML mediators. The types of mediators which 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. More details on the use of different mediators can be found in Section 2.5. The type of the mediator is reflected in the mediator specification itself and not in the reference to the mediator.

usesmediator	=	'usesMediator' idlist

An example:

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

2.3 Ontology Specification in WSML

A WSML ontology specification is identified by the **ontology** keyword optionally followed by an IRI which serves as the identifier of the ontology. In case no identifier is specified for the ontology, the locator of the ontology serves as identifier.

An example:

```
ontology family
```

An ontology specification document in WSML consists of:

ontology	=	'ontology' id? header* ontology_element*
ontology_element	=	concept
		| relation
		| instance
		| relationinstance
		| axiom

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

2.3.1 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.

Note that WSML allows inheritance of attribute definitions, which means that a concept inherits all attribute definitions of its superconcepts. In case two superconcepts have an definition for the same attribute *a*, but with a different range, these attribute definitions are interpreted conjunctively. This means that the resulting range of the attribute *a* in the subconcept is the conjunction (intersection) of the ranges of the attribute definitions in the superconcepts.

concept	=	'concept' id superconcept? nfp? attribute*

superconcept = 'subConceptOf' idlist

An example:

```
concept Human subConceptOf {Primate, LegalAgent}
  nonFunctionalProperties
    dc#description hasValue "concept of a human being"
    dc#relation hasValue humanDefinition
  endNonFunctionalProperties
  hasName ofType foaf#name
  hasParent impliesType Human
  hasChild impliesType Human
  hasAncestor impliesType Human
  hasWeight ofType xsd#float
  hasWeightInKG ofType xsd#float
  hasBirthdate ofType xsd#date
  hasObit ofType xsd#date
  hasBirthplace ofType loc#location
  isMarriedTo impliesType Human
  hasCitizenship ofType oo#country
```

WSML allows to create axioms in order to refine the definition already given in the conceptual syntax, i.e., the subconcept and attribute definitions. It is advised in the WSML specification to include the relation between the concept and the axioms related to the concept in the non-functional properties through the property dc#relation. In the example above we refer to an axiom with the identifier humanDefinition (see Section 2.3.4 for the axiom).

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. A necessary definition, such as the concept specification in the example above, specifies implications of membership of the concept for all instances of this concept. The concept description above 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, which means that if certain properties hold for the instance, the instance is inferred to be member of this concept.

WSML supports defined concepts through the use of axioms (see Section 2.3.4). The logical expression contained in the axiom 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 **memberOf** A **equivalent** ?x **memberOf** B_1 **and** ... **and** ?x **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:

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

Attributes

WSML allows two kinds of attribute definitions, namely, constraining definitions with the keyword **ofType** and inferring definitions with the keyword **impliesType**. We expect inferring attribute definitions not to be used very often if constraining definitions are allowed. However, several WSML variants, namely, WSML-Core and WSML-DL, do not allow constraining attribute definitions. In order to facilitate conceptual modeling in these language variants, we allow the use of **impliesType** in WSML.

An attribute definition of the form *A* **ofType** *D*, where *A* is an attribute identifier and *D* is a concept identifier, is a constraint on the values for attribute *A*. If the value for the attribute *A* is not known to be of type *D*, the constraint is violated and the attribute value is inconsistent with respect to the ontology. This notion of constraints corresponds with the usual database-style constraints.

The keyword **impliesType** can be used for inferring the type of a particular attribute value. An attribute definition of the form *A* **impliesType** *D*, where *A* is an attribute identifier and *D* is a concept identifier, implies membership of the concept *D* for all values of the attribute *A*. Please note that in case the range of the attribute is a datatype, the semantics of **ofType** and **impliesType** coincide, because datatypes have a known domain and thus values cannot be inferred to be of a certain datatype.

Data attributes in WSML can be distinguished from concept attributes through the meta-concept _datatype. Each datatype used in WSML is a member of this meta-concept.

Concept attributes (i.e., attributes which do not have a datatype as range) can be specified as being reflexive, transitive, symmetric, or being the inverse of another attribute, using the **reflexive**, **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.

The cardinality constraints for a single attribute are specified by including two numbers between parentheses '(' ')', indicating the minimal and maximal cardinality, after the **ofType** (or **impliesType**) 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.

```
attribute          = [attr]: id attributefeature* att_type cardinality? [type]: id nfp?
att_type           = 'ofType'
                   | 'impliesType'
cardinality        = '(' [min_cardinality]: digit+ [max_cardinality]: cardinality_number? ')'
cardinality_number = {finite_cardinality} digit+
                   | {infinite_cardinality} '*'
attributefeature   = 'transitive'
                   | 'symmetric'
                   | 'inverseOf' '(' id ')'
                   | 'reflexive'
```

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*.

An example of a concept definition with attribute definitions:

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

2.3.2 Relations

A relation definition starts with the **relation** keyword, which is optionally followed by the identifier of the relation. WSML allows the specification of relations with arbitrary arity. The domain of the parameters can be optionally specified using the keyword **impliesType** or **ofType**. Note that parameters of a relation are strictly ordered. A relation definition is optional completed by the keyword **subRelationOf** followed by one or more identifiers of superrelations. Finally an optional **nonFunctionalProperties** block can be specified.

Relations in WSML can have an arbitrary arity and values for the parameters can be constrained using parameter type definitions of the form (**ofType** *type-name*) and (**impliesType** *type-name*). The definition of relations requires either the indication of the arity or of the parameter parameter definitions. The usage of **ofType** and **impliesType** correspond with the usage in attribute definitions. Namely, parameter definitions with the **ofType** keyword are used to constrain the allowed parameter values, whereas parameter definitions with the **impliesType** keyword are used to infer concept membership of parameter values.

relation	=	'relation' id arity? paramtyping? superrelation? nfp?
arity	=	'/' pos_integer
paramtyping	=	'(' paramtype moreparamtype* ')'
paramtype	=	att_type idlist
moreparamtype	=	',' paramtype
superrelation	=	'subRelationOf' idlist

Two examples, one with parameter definitions and one with an arity definition:

```
relation distance (ofType City, ofType City, impliesType xsd#decimal) subRelationOf measurement
```

```
relation distance/3
```

Similarly as for concepts, the exact meaning of a relation can be defined using axioms. For example one could axiomatize the transitive closure for a property or further restrict the domain of one of the parameters. As with concepts, it is recommended to indicate related axioms using the non-functional property dc#relation.

2.3.3 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. 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.

instance	=	'instance' id? memberof? nfp? attributevalue*
memberof	=	'memberOf' idlist
attributevalue	=	id 'hasValue' valuelist

An example:

```
instance Mary memberOf {Parent, Woman}
  nfp
    dc#description hasValue "Mary is parent of the twins Paul and Susan"
  endnfp
  hasName hasValue "Maria Smith"
  hasBirthdate hasValue _date(1949,9,12)
  hasChild hasValue {Paul, Susan}
```

Instances explicitly specified in an ontology are those which are shared together as part of the ontology. However, most instance data exists outside the ontology in private data stores. Access to 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 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).

relationinstance	=	'relationInstance' [name]: id? [relation]: id '(' value (',' value)* ')' nfp?

An example of an instance of a ternary relation (remember that the identifier is optional, see also Section 2.1.2):

```
relationInstance distance(Innsbruck, Munich, 234)
```

2.3.4 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 logical expression must be followed by either a blank or a new line. The language allowed for the logical expression is explained in Section 2.8.

```
axiom           =  'axiom'  axiomdefinition
axiomdefinition =  id
                 | id? nfp? log_definition
log_definition  =  'definedBy'  log_expr
```

An example of a defining axiom:

```
axiom humanDefinition
  definedBy
    ?x memberOf Human equivalent
    ?x memberOf Animal and
    ?x memberOf LegalAgent.
```

WSML allows to specify database-style constraints. An example of a constraining axiom:

```
axiom humanBMIConstraint
  definedBy
    !- naf bodyMassIndex(bmi hasValue ?b, length hasValue ?l, weight hasValue ?w)
      and ?x memberOf Human and
      ?x[length hasValue ?l,
        weight hasValue ?w,
        bmi hasValue ?b].
```

2.4 Capability and Interface Specification in WSML

The desired and provided functionality are described in WSML in the form of capabilities. The desired capability is part of a goal and the provided capability is part of a web service. The interaction style of both the requester and the provider are described in interfaces, as part of the goal and the web service, respectively.

2.4.1 Capabilities

A WSML goal or web service may only have one capability. The specification of a capability is optional.

A capability description starts with the **capability** keyword, (optionally) followed by the name (identifier) of the capability. This is followed by an optional **nonFunctionalProperties** block, an optional **importsOntology** block and an optional **usesMediator** block. The **sharedVariables** block is used to indicate the variables which are shared between the preconditions, postconditions, assumptions and effects of the capability, which are defined in the **precondition**, **postcondition**, **assumption**, and **effect** definitions, respectively. The number of such definitions is not restricted. Each of these definitions consists of the keyword, an optional identifier, an optional **nonFunctionalProperties** block and a logical expression preceded by the **definedBy** keyword, and thus has the same content as an axiom (see Section 2.3.4). The language allowed for the logical expression differs per WSML variant and is explained in the respective chapters.

```
capability       =  'capability'  id?  header*  sharedvardef?  pre_post_ass_or_eff*
sharedvardef     =  'sharedVariables'  variablelist
pre_post_ass_or_eff =  'precondition'  axiomdefinition
                     | 'postcondition'  axiomdefinition
                     | 'assumption'  axiomdefinition
                     | 'effect'  axiomdefinition
```

An example of a capability specified in WSML:

```
capability
  sharedVariables ?child
  precondition
    nonFunctionalProperties
        dc#description hasValue "The input has to be boy or a girl
          with birthdate in the past and be born in Germany."
    endNonFunctionalProperties
    definedBy
      ?child memberOf Child
          and ?child[hasBirthdate hasValue ?brithdate]
          and wsml#dateLessThan(?birthdate,wsml#currentDate())
          and ?child[hasBirthplace hasValue ?location]
          and ?location[locatedIn hasValue oo#de]
          or (?child[hasParent hasValue ?parent] and
                ?parent[hasCitizenship hasValue oo#de] ) .

  assumption
      nonFunctionalProperties
        dc#description hasValue "The child is not dead"
```

```
        endNonFunctionalProperties
        definedBy
           ?child memberOf Child
                 and naf ?child[hasObit hasValue ?x].

    effect
        nonFunctionalProperties
           dc#description hasValue "After the registration the child
                 is a German citizen"
        endNonFunctionalProperties
        definedBy
           ?child memberOf Child
                 and ?child[hasCitizenship hasValue oo#de].
```

2.4.2 Interface

A WSML goal may request multiple interfaces and a web service may offer multiple interfaces. The specification of an interface is optional.

An interface specification starts with the **interface** keyword, (optionally) followed by the name (identifier) of the interface. This is followed by an optional **nonFunctionalProperties** block, an optional **importsOntology** block and an optional **usedMediator** block and then by an optional choreography block consisting of the keyword **choreography** followed by the identifier of the choreography and an optional orchestration block consisting of the keyword **orchestration** followed by the identifier of the orchestration. Notice that thus an interface can have at most one choreography and at most one orchestration. It is furthermore possible to reference interface which have been specified at a different location. For reasons of convenience, WSML allows to reference multiple interfaces using an argument list.

```
interfaces     =  interface
                | minterfaces
minterfaces    =  'interface'  '{'  id  moreids*  '}'
interface      =  'interface'  id?  header*  choreography?  orchestration?
choreography   =  'choreography'  id
orchestration  =  'orchestration'  id
```

An example of an interface and an example of references to multiple interfaces:

```
  interface
   choreography _"http://example.org/mychoreography"
   orchestration _"http://example.org/myorchestration"

   interface {_"http://example.org/mychoreography", _"http://example.org/mychoreography"}
```

We do not define ways to specify the choreography and orchestration here. Instead, we refer the reader to the corresponding WSMO deliverables D14 [Roman & Scicluna, 2005a] and D15 [Roman & Scicluna, 2005b].

2.5 Goal Specification in WSML

A WSML goal specification is identified by the **goal** keyword optionally followed by an IRI which serves as the identifier of the goal. In case no identifier is specified for the goal, the locator of the goal serves as identifier.

An example:

```
 goal _"http://example.org/Germany/GetCitizenShip"
```

A goal specification document in WSML consists of:

```
goal =  'goal'  id?  header*  capability?  interfaces*
```

The elements of a goal are the capability and the interfaces which are explained in the previous section.

2.6 Mediator Specification in WSML

WSML allows for the specification of four kinds of mediators, namely ontology mediators, mediators between web services, mediators between goals and mediators between web services and goals. These mediators are referred via the keywords **ooMediator**, **wwMediator**, **ggMediator** and **wgMediator**, respectively (cf. [Roman et al., 2004]).

```
mediator =  oomediator
```

```
                | ggmediator
                | wgmediator
                | wwmediator
```

A WSML mediator specification is identified by the keyword indicating a particular kind of mediator (**ooMediator**, **wwMediator**, **ggMediator**, **wgMediator**), optionally followed by an IRI which serves as the identifier of the mediator. When no identifier is specified for the mediator, the locator of the mediator serves as identifier.

An example:

```
ooMediator _"http://example.org/ooMediator"
```

All types of mediators share the same syntax for the sources, targets and used services:

```
use_service = 'usesService' id
source      = 'source' id
msources    = 'source' '{ id ( ',' id )* '}'
sources     =  source
             | msources
target      = 'target' id
```

2.6.1 ooMediators

ooMediators are used to connect ontologies to other ontologies, web services, goals and mediators. ooMediators take care of resolving any occurring heterogeneity.

The **source** of an ooMediator in WSML may only contain identifiers of ontologies and other ooMediators as source.

An ooMediator in WSML may only have one **target**. The target may be the identifier of an ontology, a goal, a web service or another mediator.

The keyword **usesService** is used to identify a goal which declaratively describes the mediation service, a web service which actually implements the mediation or a wwMediator which links to such a web service. The entity pointed to is given by an identifier

```
oomediator = 'ooMediator' id? nfp? importedontology? sources target? use_service?
```

An **ooMediator** is used to import (parts of) ontologies and resolve heterogeneity. This concept of mediation between ontologies is more flexible than the **importsOntology** statement, which is used to import a WSML ontology into another WSML specification. The ontology import mechanism appends the definitions in the imported ontology to the importing specification.

In fact, importing ontologies can be seen as a simple form of mediation, in which no heterogeneity is resolved. However, in the general case there are mismatches and overlaps between the different ontologies which require mediation. Furthermore, if the imported ontology is specified using a WSML variant which has an undesirable expressiveness, a mediator could be used to weaken the definitions to the desired expressiveness.

2.6.2 wwMediators

wwMediators connect Web Services, resolving any data, process and protocol heterogeneity between the two.

A wwMediators in WSML may only have one **source**. The source may be the identifier of a web service or another wwMediator.

A wwMediators in WSML may only have one **target**. The target may be the identifier of a web service or another wwMediator.

```
wwmediator = 'wwMediator' id? header* source? target? use_service?
```

2.6.3 ggMediators

ggMediators connect different goals, enabling goals to refine more general goals and thus enabling reuse of goal definitions.

A ggMediators in WSML may only have one **source**. The source may be the identifier of a goal or another ggMediator.

A ggMediators in WSML may only have one **target**. The target may be the identifier of a goal or another ggMediator.

ggmediator = 'ggMediator' id? header* source* target? use_service?

2.6.4 wgMediators

wgMediators connect goals and web services, resolving any data, process and protocol heterogeneity.

A wgMediators in WSML may only have one **source**. The source may be the identifier of a web service or another wgMediator.

A wgMediators in WSML may only have one **target**. The target may be the identifier of a goal or a ggMediator.

wgmediator = 'wgMediator' id? header* source? target? use_service?

By externalizing the mediation services from the implementation of ontologies, goals and web services, WSML allows loose coupling of elements; the mediator is responsible for relating the different elements to each other and resolving conflicts and mismatches. For more details we refer to [Roman et al., 2004].

None of the elements in a mediator has any meaning in the logical language. In fact, the complexity of a mediator is hidden in the actual description of the mediator. Instead, the complexity is either in the implementation of the mediation service, in which case WSML does not support the description because WSML is only concerned with the interface description, or in the functional description of the web service or the goal which is used to specify the desired mediation service. As discussed in [Keller et al., 2004], these descriptions often need a very expressive language.

2.7 Web Service Specification in WSML

A WSML web service specification is identified by the **webService** keyword optionally followed by an IRI which serves as the identifier of the web service. If no identifier is specified for the web service, the locator of the web service specification serves as identifier.

A web service specification document in WSML consists of:

webservice = 'webService' id? header* capability? interface*

An example:

```
webService _"http://example.org/Germany/BirthRegistration"
```

The elements of a web service are capability and interface which are explained in Section 2.4.

2.8 Logical Expressions in WSML

Logical expressions occur within axioms and the capabilities which are specified in the descriptions of goals and Semantic Web services. In the following, we give a syntax specification for general logical expressions in WSML. The general logical expression syntax presented in this chapter encompasses all WSML variants and is thus equivalent to the WSML-Full logical expression syntax. In the subsequent chapters, we specify for each of the WSML variants the restrictions the variant poses on the logical expression syntax.

In order to specify the WSML logical expressions, we introduce a new kind of identifier: variables.

Variables

Variable names start with an initial question mark, "?". Variables may occur in place of concepts, attributes, instances, relation arguments or attribute values. 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 implicitly universally quantified outside the formula, unless the formula is part of a capability description and the variable is explicitly mentioned in the **sharedVariables** block.

variable = '?' alphanum+

Examples of variables are: ?x, ?y1, ?myVariable

The syntax specified in the following is inspired by First-Order Logic [Enderton, 2002] and F-Logic [Kifer et al., 1995].

We start with the definition of the basic vocabulary for building logical expressions. Then, we define how the elements of the basic vocabulary can be composed in order to obtain admissible logical expressions. Definition 2.1 defines the notion of a vocabulary *V* of a WSML language *L*.

Definition 2.1. A vocabulary *V* of a WSML language *L(V)* consists of the following:

- A set of identifiers V_{ID}.
- A set of object constructors $V_O \subseteq V_{ID}$.
- A set of function symbols $V_F \subseteq V_O$.
- A set of datatype wrappers $V_D \subseteq V_O$.
- A set of data values $V_{DV} \subseteq V_O$ which encompasses all string, integer and decimal values.
- A set of anonymous identifiers $V_A \subseteq V_O$ of the form _#, _#1, _#2, etc....
- A set of relation identifiers $V_R \subseteq V_{ID}$.
- A set of variable identifiers $V_V \subseteq V_{ID}$ of the form ?*X*.

WSML allows the following logical connectives: **and**, **or**, **implies**, **impliedBy**, **equivalent**, **neg**, **naf**, **forAll** and **exists** and the following auxiliary symbols: '(', ')', '[', ']', ',', '**=**', '**!=**', '**:=:**', **memberOf**, **hasValue**, **subConceptOf**, **ofType**, and **impliesType**. Furthermore, WSML allows use of the symbol '**:-**' for Logic Programming rules and the use of the symbol '**!-**' for database-style constraints.

Definition 2.2 defines the set of terms *Term(V)* for a given vocabulary *V*.

Definition 2.2. Given a vocabulary *V*, the set of terms *Term(V)* in WSML is defined as follows:

- Any $f \in V_O$ is a term.
- Any $v \in V_V$ is a term
- If $f \in V_F$ and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.
- If $f \in V_D$ and $dv_1, ..., dv_n$ are in $V_{DV} \cup V_V$, then $f(dv_1, ..., dv_n)$ is a term.

As usual, the set of ground terms *GroundTerm(V)* is the maximal subset of *Term(V)* which does not contain variables.

Based on the basic constructs of logical expressions, the terms, we can now define formulae. In WSML, we have atomic formulae and complex formulae. Each formula is terminated by a period.

Definition 2.3. Given a set of terms *Term(V)*, the set of atomic formulae in *L(V)* is defined by:

- If $r \in V_R$ and $t_1, ..., t_n$ are terms, then $r(t_1, ..., t_n)$ is an atomic formula in *L(V)*.
- If $\alpha, \beta \in Term(V)$ then α **=** β, α **:=:** β and α **!=** β are atomic formulae in *L(V)*.
- If $\alpha, \beta \in Term(V)$ and $\gamma \in Term(V)$ or γ is of the form $\{ \gamma_1,...,\gamma_n \}$, with $\gamma_1,...,\gamma_n \in Term(V)$, then:
  - α **subConceptOf** γ is an atomic formula in *L(V)*
  - α **memberOf** γ is an atomic formula in *L(V)*
  - α[β **ofType** γ] is an atomic formula in *L(V)*
  - α[β **impliesType** γ] is an atomic formula in *L(V)*
  - α[β **hasValue** γ] is an atomic formula in *L(V)*

Given the atomic formulae, we recursively define the set of formulae in *L(V)* in definition 2.4.

Definition 2.4. The set of formulae in *L(V)* is defined by:

- Every atomic formula in *L(V)* is a formula in *L(V)*.
- Let α, β be formulae which do not contain the symbols '**:-**' and '**!-**', and let $?x_1,...,?x_n$ be variables, then:
  - α **and** β is a formula in *L(V)*.
  - α **or** β is a formula in *L(V)*.
  - **neg** α is a formula in *L(V)*.
  - **naf** α is a formula in *L(V)*.
  - **forall** $?x_1,...,?x_n$ (α) is a formula in *L(V)*.
  - **exists** $?x_1,...,?x_n$ (α) is a formula in *L(V)*.
  - α **implies** β is a formula in *L(V)*.
  - α **impliedBy** β is a formula in *L(V)*.

- *α* **equivalent** *β* is a formula in *L*(*V*).
- *α* **:-** *β* is a formula in *L*(*V*). This formula is called an *LP (Logic Programming) rule*. *α* is called the *head* and *β* is called the *body* of the rule.
- **!-** *α* is a formula in *L*(*V*). This formula is called a *constraint*. We say *α* is a constraint of the knowledge base.

Note that WSML allows the symbols ->, <- and <-> as synonyms for **implies**, **impliedBy**, and **equivalent**, respectively.

The precedence of the operators is as follows: **implies, equivalent, impliedBy < or, and < neg, naf**. Here, op_1 < op_2 means that operator op_2 binds stronger than operator op_1. The precedence prevents extensive use of parenthesis and thus helps to achieve a better readability of logical expressions.

Any formula followed by a dot '.' is a WSML logical expression.

To enhance the readability of logical expressions it is possible to abbreviate a conjunction of several modules with the same subject as one compound molecule. E.g., the three molecules

```
Human subConceptOf Mammal
  and Human[hasName ofType foaf:name] and Human[hasChild impliesType Human]
```

can be written as

```
Human[hasName ofType foaf:name, hasChild impliesType Human] subConceptOf Mammal
```

The following are examples of WSML logical expressions (note that variables are implicitly universally quantified):

No human can be both male and female:

```
!- ?x[gender hasValue {?y, ?z}] memberOf Human and ?y = Male and ?z = Female.
```

A human who is not a man is a woman:

```
?x[gender hasValue Woman] impliedBy neg ?x[gender hasValue Man].
```

The brother of a parent is an uncle:

```
?x[uncle hasValue ?z] impliedBy ?x[parent hasValue ?y] and ?y[brother hasValue ?z].
```

Do not trust strangers:

```
?x[distrust hasValue ?y] :- naf ?x[knows hasValue ?y].
```

A complex example:

```
f(?x)[a hasValue ?y] impliedBy g(?x) memberOf b :- naf ?x[c hasValue ?y] and p(?x, ?z) or exists ?w (h(?w) :=: ?z and q(f(f(f(?w))),h(?x))).
```

3 WSML-Core

As described in the introduction to this Part, there are several WSML language variants with different underlying logical formalisms. The two main logical formalisms exploited in the different WSML language variants are Description Logics [Baader et al., 2003] (exploited in WSML-DL) and Rule Languages [Lloyd, 1987] (exploited in WSML-Flight and WSML-Rule). WSML-Core, which is described in this chapter, marks the intersection of both formalisms. WSML-Full, which is the union of both paradigms, is described in Chapter 7.

WSML-Core is based on the Logic Programming subset of Description Logics described in [Grosof et al., 2003]. More specifically, WSML-Core is based on plain (function- and negation-free) Datalog, thus, 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 polynomial time.[2]

Many of the syntactical restrictions imposed by WSML-Core are a consequence of the limitation of WSML-Core to Description Logic Programs as defined in [Grosof et al., 2003].

This chapter is further structured as follows. We first introduce basics of the WSML-Core syntax, such as the use of namespaces, identifiers, etc. in Section 3.1. We describe the restrictions WSML-Core poses on the modeling of ontologies, goals, mediators and web services in sections 3.2, 3.3, 3.4 and 3.5, respectively. Finally, we describe the restrictions on logical expressions in WSML-Core in Section 3.6.

3.1 Basic WSML-Core Syntax

WSML-Core inherits the basics of the WSML syntax specified in Section 2.1. 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 identifiers. The vocabulary of WSML-Core is *separated* similarly to OWL DL.

Definition 3.1. A WSML-Core vocabulary V follows the following restrictions:

- V_C, V_D, V_R, V_I and V_{NFP} are the sets of concept, datatype, relation, instance and non-functional property identifiers. These sets are all subsets of the set of IRIs and are pairwise disjoint.
- The set of attribute names is equivalent to V_R
- The set of relation identifiers V_R is split into two disjoint sets, V_{RA} and V_{RC}, which correspond to relations with an abstract and relations with a concrete range, respectively.

3.2. WSML-Core Ontologies

In this section we explain the restrictions on the WSML ontology modeling elements imposed by WSML-Core. The restrictions posed on the conceptual syntax for ontologies is necessary because of the restriction imposed on WSML-Core by the chosen underlying logical formalism (the intersection of Datalog and Description Logics), cf. [de Bruijn et al., 2004].

The grammar fragments shown in the following subsections only concern those parts of the grammar which are different from the general WSML grammar.

3.2.1 Concepts

WSML-Core poses a number of restrictions on attribute definitions. Most of these restrictions stem from the fact that it is not possible to express constraints in WSML-Core, other than for datatypes.

WSML-Core does not allow for the specification of the attribute features **reflexive**, **transitive**, **symmetric** and **inverseOf**. This restriction stems from the fact that reflexivity, transitivity, symmetricity and inversity of attributes are defined locally to a concept in WSML as opposed to Description Logics or OWL. You can however define global transitivity, symmetricity and inversity of attributes just like in DLs or OWL by defining respective axioms (cf. Definition 3.3 below).

Cardinality constraints are not allowed and thus it is not possible to specify functional properties.

It is not allowed to specify constraining attribute definitions, other than for datatype ranges. In other words, attribute definitions of the form: *A* **ofType** *D* are not allowed, unless *D* is a datatype identifier.

```
attribute = [attr]: id att_type [type]: id nfp?
```

3.2.2 Relations

In WSML-Core, the arity of relations is restricted to two. The domain of the two parameters may be given using the keyword **impliesType** or **ofType**. However, the **ofType** keyword is only allowed in combination with a datatype and only the second parameter may have a datatype as its range.

relation	=	'relation' id '/2'? paramtyping? superrelation? nfp?
paramtyping	=	'(' 'impliesType' idlist ',' att_type idlist ')'
superrelation	=	'subRelationOf' idlist

Binary relations are in an ontological sense nothing more than attribute definitions. In most cases, it is thus highly recommended to define attributes on concepts instead of using binary relations.

3.2.3 Instances

WSML-Core does not impose restrictions on the specification of instances for concepts. Relation instances are only allowed for binary relations. Both values of the relation have to be specified and have to correspond to its signature, this includes that the first value must not be an data value.

3.2.4 Axioms

WSML-Core does not impose restrictions on the specification of axioms, besides the fact that WSML-Core only allows the use of a restricted form of the WSML logical expression syntax. These restrictions are specified in the Section 3.6.

3.3. Goals in WSML-Core

Goals in WSML-Core follow the common WSML syntax. The logical expressions in the postconditions and effects are limited to WSML-Core logical expressions.

3.4. Mediators in WSML-Core

Mediators in WSML-Core follow the common WSML syntax.

3.5. Web Services in WSML-Core

Web Services in WSML-Core follow the common WSML syntax. The logical expressions in the assumptions, preconditions, effects and postconditions are limited to WSML-Core logical expressions.

3.6. WSML-Core Logical Expression Syntax

WSML-Core allows only a restricted form of logical expressions. There are two sources for these restrictions. Namely, the restriction of the language to a subset of Description Logics restricts the kind of formulas which can be written down to the two-variable fragment of first-order logic. Furthermore, it disallows the use of function symbols and restricts the arity of predicates to unary and binary and chaining variables over predicates. The restriction of the language to a subset of Datalog (without equality) disallows the use of the equality symbol, disjunction in the head of a rule and existentially quantified variables in the head of the rule.

Formulae in WSML Core logical expressions are, compared to Definitions 2.5 and 2.6 restricted to **conjunctions of WSML-Core clauses**. These WSML-Core clauses are defined in the following.

Let *V* be a WSML-Core vocabulary. Let further $\gamma \in V_C$, Γ be either an identifier in V_C or a list of identifiers in V_C, Δ be either an identifier in V_D or a list of identifiers in V_D, $\varphi \in V_I$, ψ be either an identifier in V_I or a list of identifiers in V_I, $p,q \in V_{RA}$, $s,t \in V_{RC}$, and *Val* be either a data value or a list of data values.

Definition 3.2. The set of atomic formulae in *L(V)* is defined as follows:

- γ **subConceptOf** Γ is an atomic formula in *L(V)*
- φ **memberOf** Γ is an atomic formula in *L(V)*
- φ[*p* **ofType** Δ] is an atomic formula in *L(V)*
- φ[*s* **impliesType** Δ] is an atomic formula in *L(V)*
- φ[*s* **impliesType** Γ] is an atomic formula in *L(V)*
- φ[*p* **hasValue** ψ] is an atomic formula in *L(V)*
- φ[*s* **hasValue** *Val*] is an atomic formula in *L(V)*

These are the only atomic clauses allowed in WSML Core, i.e., compared with general logical expressions, WSML core only allows *ground* facts.

Let Var_1, Var_2, ... be arbitrary WSML variables. We call molecules of the form Var_i **memberOf** Γ *a-molecules*, and molecules of the forms , Var_i[*p* **hasValue** Var_k] and Var_i[*p* **hasValue** {Var_{k1}, Var_{kl}}] *b-molecules*, respectively.

In the following, *F* stands for an lhs-formula, with the set of lhs-formulae defines as follows:

- Any b-molecule is an lhs-formula
- if F_1 and F_2 are lhs-formulae, then F_1 **and** F_2 is an lhs-formula
- if F_1 and F_2 are lhs-formulae, then F_1 **or** F_2 is an lhs-formula

G,H stand for arbitrary WSML logical expressions formed by a-molecules and the operator **and**.

In the following, *G,H* stand for rhs-formulae, with the set of rhs-formulae defines as follows:

- Any a-molecule is an rhs-formula
- if *G* and *H* are lhs-formulae, then *G* **and** *H* is an rhs-formula

Definition 3.3. The set of WSML-Core formulae is defined as follows:

- Any atomic formula is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_1[*p* hasValue Var_3] **and** Var_3[*p* hasValue Var_2] (globally transitive attribute/relation) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_2[*p* hasValue Var_1] (globally symmetric attribute/relation) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_1[*q* hasValue Var_2] (globally sub-attribute/relation)) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_2[*q* hasValue Var_1] (globally inverse attribute/relation)) is a formula in *L(V)*.
- *G* **equivalent** *H* is a formula in *L(V)* if it contains only one WSML variable.
- *H* **impliedBy** *F* is a formula in *L(V)* if all the WSML variables occurring in H occur in F as well and the *variable graph* of *F* is connected and acyclic.
- If *Head* **impliedBy** *Body* is a formula in *L(V)*, then *Body* **implies** *Head* a formula in *L(V)* .
- Any occurrence of a molecule of the form Var_1[*p* **hasValue** Var_2] in a WSML-Core clause can be interchanged with *p*(Var_1,Var_2) (i.e., these two forms can be used interchangably in WSML Core).

Here, the *variable graph* of a logical expression E is defined as the undirected graph having all WSML variables in E as nodes and an edge from Var_2 to Var_2 for every molecule Var_1[*p* **hasValue** Var_2], or *p*(Var_1,Var_1), respectively.

Note that wherever an a-molecule (or b-molecule, respectively) is allowed in a WSML-Core clause, also compound molecules abbreviating conjunctions of a-molecules (or b-molecules, respectively), as mentioned in the end of Section 2.8 above, are allowed.

4. WSML-DL

WSML-DL will be an extension of WSML-Core to a full-fledged description logic with an expressiveness similar to OWL DL. WSML-DL 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-DL specification. Furthermore, all consequences inferred from a WSML-Core specification are also valid consequences of the same specification in WSML-DL. Finally, if a WSML-DL specification falls inside the WSML-Core fragment then all consequences wrt. the WSML-DL semantics also hold wrt. the WSML-Core semantics.

It is planned for WSML-DL to allow an alternate syntax for logical expressions in order to better reflect the DL-based modeling style, since all DL expressions follow more-or-less the same patterns when writing them down using FOL syntax.

4.1 Basic WSML-DL Syntax

4.2. WSML-DL Ontologies

4.3. Goals in WSML-DL

4.4. Mediators in WSML-DL

4.5. Web Services in WSML-DL

4.6. WSML-DL Logical Expression Syntax

4.7. Differences between WSML-Core and WSML-DL

WSML-DL extends WSML-Core to a full-fledged Description Logic with extensive datatype support based on OWL-E [Pan and Horrocks, 2004].

5. WSML-Flight

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.

WSML-Flight adds the following features to WSML-Core:

- N-ary relations with arbitrary parameters
- Constraining attribute definitions for the abstract domain
- Cardinality constraints
- (Locally Stratified) default negation in logical expressions (in the body of the rule)
- Expressive logical expressions, namely, the full Datalog subset of F-Logic, extended with inequality (in the body) and locally stratified negation
- Meta-modeling. WSML-Flight no longer requires a separation of vocabulary (wrt. concepts, instances, relations)

Default negation means that the negation of a fact is true, unless the fact is *known* to be true. Locally stratified negation means that the definition of a particular predicate does not negatively depend on itself.

This chapter is further structured as follows. We first introduce basics of the WSML-Flight syntax in Section 5.1. We describe the restrictions WSML-Flight poses on the modeling of ontologies, goals, mediators and web services in sections 5.2, 5.3, 5.4 and 5.5, respectively. Finally, we describe the restrictions on logical expressions in WSML-Flight in Section 5.6.

5.1 Basic WSML-Flight Syntax

WSML-Flight adheres to the WSML syntax basics described in Section 2.1. The restrictions posed on this basic syntax by WSML-Core do not apply to WSML-Flight.

5.2. WSML-Flight Ontologies

Compared to WSML-Core, WSML-Flight does allow additional functionality for attribute definitions, relations, functions, and relation instances. In fact, the conceptual syntax for ontology modeling completely corresponds with the ontology modeling elements introduced in section 2.3.

Note that for axioms, we only allow a restricted form of logical expressions, as defined in Section 5.6.

5.3. Goals in WSML-Flight

Goals in WSML-Flight follow the common WSML syntax. The logical expressions in the postconditions and effects are limited to WSML-Flight logical expressions.

5.4. Mediators in WSML-Flight

Mediators in WSML-Core follow the common WSML syntax.

5.5. Web Services in WSML-Flight

Web Services in WSML-Flight follow the common WSML syntax. The logical expressions in the assumptions, preconditions, effects and postconditions are limited to WSML-Flight logical expressions.

5.6. WSML-Flight Logical Expression Syntax

WSML-Flight is a rule language based on the Datalog subset of F-Logic, extended with locally stratified default negation, the inequality symbol '!=' and the unification operator '='. Furthermore, WSML-Flight allows monotonic Lloyd-Topor [Lloyd and Topor, 1984], which means that we allow classical implication and conjunction in the head of a rule and we allow disjunction in the body of a rule.

The head and the body of a rule are separated using the Logic Programming implication symbol '**:-**'. This additional symbols is required because negation-as-failure (**naf**) is not defined for classical implication (**implies**, **impliedBy**). WSML-Flight allows classical implication in the head of the rule. Consequently, every WSML-Core logical expression is a WSML-Flight rule with an empty body.

The syntax for logical expressions of WSML Flight is the same as described in Section 2.8 with the restrictions

described in the following. We define the notion of a WSML-Flight vocabulary in Definition 5.1.

Definition 5.1. Any WSML vocabulary (see Definition 2.1) is a WSML-Flight vocabulary.

Definition 5.2 defines the set of WSML-Flight terms $Term_{Flight}(V)$ for a given vocabulary V.

Definition 5.2. Given a vocabulary V, the set of terms $Term_{Flight}(V)$ in WSML-Flight is defined as follows:

- Any $f \in V_{OC}$ is a term.
- Any $v \in V_V$ is a term
- If $d \in V_D$ and $dv_1, ..., dv_n$ are in $V_{DV} \cup V_V$, then $d(dv_1, ..., dv_n)$ is a term.

As usual, the set of ground terms $GroundTerm_{Flight}(V)$ is the maximal subset of $Term_{Flight}(V)$ which does not contain variables.

WSML-Flight does not allow the equality operator (**:=:**). Therefore, the set of admissible atomic formulae in WSML-Flight does not contain α **:=:** β for terms α, β.

Definition 5.3. Given a set of WSML-Flight terms $Term_{Flight}(V)$, an atomic formula in $L(V)$ is defined by:

- If $r \in V_R$ and $t_1, ..., t_n$ are terms, then $r(t_1, ..., t_n)$ is an atomic formula in $L(V)$.
- If $\alpha, \beta \in Term_{Flight}(V)$ then α **=** β, and α **!=** β are atomic formulae in $L(V)$.
- If $\alpha, \beta \in Term_{Flight}(V)$ and $\gamma \in Term(V)$ or γ is of the form { $\gamma_1,...,\gamma_n$ } with $\gamma_1,...,\gamma_n \in Term_{Flight}(V)$, then:
  - α **subConceptOf** γ is an atomic formula in $L(V)$
  - α **memberOf** γ is an atomic formula in $L(V)$
  - α[β **ofType** γ] is an atomic formula in $L(V)$
  - α[β **impliesType** γ] is an atomic formula in $L(V)$
  - α[β **hasValue** γ] is an atomic formula in $L(V)$

A ground atomic formula is an atomic formula with no variables.

Definition 5.4. Given a WSML-Flight vocabulary V, the set of formulae in $L(V)$ is recursively defined as follows:

- Given a head-formula $\beta \in Head(V)$ and a body-formula $\alpha \in Body(V)$, β **:-** α is a formula. Here we call α the *body* and β the *head* of the formula. The formula is admissible if (1) α is an admissible body formula, (2) β is an admissible head formula, (3) the safety condition holds and (4) the resulting WSML-Flight knowledge base is *locally stratified*.
- We define the set of admissible head formula $Head(V)$ as follows:
  - Any atomic formula α which does not contain the inequality symbol (!=) or the unification operator (=) is in $Head(V)$.
  - Let $\alpha,\beta \in Head(V)$, then α **and** β is in $Head(V)$.
  - α **implies** β is in $Head(V)$.
  - α **impliedBy** β is in $Head(V)$.
  - α **equivalent** β is in $Head(V)$.
- Any admissible head formula in $Head(V)$ is a formula in $L(V)$.
- We define the set of admissible body formulae $Body(V)$ as follows:
  - Any atomic formula α is in $Body(V)$
  - For $\alpha \in Body(V)$, **naf** α is in $Body(V)$.
  - For $\alpha,\beta \in Body(V)$, α **and** β is in $Body(V)$.
  - For $\alpha,\beta \in Body(V)$, α **or** β is in $Body(V)$.
- Any formula of the form !- α with $\alpha \in Body(V)$ is an admissible formula and is called a *constraint*.

As with the general WSML logical expression syntax, <-, -> and <-> can be seen as synonyms of the keywords **inplies**, **impliedBy** and **equivalent**, respectively.

In order to check the *safety condition* for a WSML-Flight rule, the following transformations should be applied until no transformation rule is applicable:

- Rules of the form A_1 **and** ... **and** A_n **:-** B are split into n different rules:
  - A_1 **:-** B
  - ...
  - A_n **:-** B
- Rules of the form A_1 **equivalent** A_2 **:-** B split into 2 rules:
  - A_1 **implies** A_2 **:-** B
  - A_1 **impliedBy** A_2 **:-** B
- Rules of the form A_1 **impliedBy** A_2 **:-** B are transformed to:
  - A_1 **:-** A_2 **and** B

- Rules of the form A_1 **implies** A_2 **:-** *B* are transformed to:
  - A_2 **:-** A_1 **and** *B*
- Rules of the form *A* **:-** B_1 **or** ... **or** B_n are split into *n* different rules:
  - *A* **:-** B_1
  - ...
  - *A* **:-** B_n

Application of these transformation rules yields a set of WSML-Flight rules with only one atomic formula in the head and a conjunction of literals in the body.

The safety condition holds for a WSML-Flight rule if every variable which occurs in the rule occurs in a positive body literal which does not correspond to a built-in predicate. For example, the following rules are not safe and thus not allowed in WSML-Flight:

```
p(?x) :- q(?y).
a[b hasValue ?x] :- ?x > 25.
?x[gender hasValue male] :- naf ?x[gender hasValue female].
```

We require each WSML-Flight knowledge base to be *locally stratified*. Appendix A of [Kifer et al., 1995] explains local stratification for a frame-based logical language.

The following are examples of WSML-Flight logical expressions:

```
?y memberOf ?z impliedBy ?z memberOf ?x :- naf ?y[a hasValue ?x, start hasValue _date(2005,6,6,0,0), nr hasValue 10, name hasValue "myName"] and p
```

5.7. 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, Constraining attribute definitions for the abstract domain, Cardinality constraints, (locally stratified) default negation in logical expressions, (in)equality in the logical language (in the body of the rule), Full-fledged rule language (based on the Datalog subset of F-Logic).

6. WSML-Rule

WSML-Rule is an extension of WSML-Flight in the direction of Logic Programming. WSML-Rule no longer requires safety of rules and allows the use of function symbols. The only differences between WSML-Rule and WSML-Flight are in the logical expression syntax.

WSML-Rule is both syntactically and semantically layered on top of WSML-Flight and thus each valid WSML-Flight specification is a valid WSML-Rule specification. Because the only differences between WSML-Flight and WSML-Rule are in the logical expression syntax, we do not explain the conceptual syntax for WSML-Rule.

Section 6.1 defines the logical expression syntax of WSML-Rule. Section 6.2 outlines the differences between WSML-Flight and WSML-Rule.

6.1. WSML-Rule Logical Expression Syntax

WSML-Rule is a simple extension of WSML-Flight. WSML-Rule allows the unrestricted use of function symbols and no longer requires the safety condition, i.e., variables which occur in the head are not required to occur in the body of the rule.

The syntax for logical expressions of WSML Rule is the same as described in Section 2.8 with the restrictions which are described in the following. We define the notion of a WSML-Rule vocabulary in Definition 6.1.

Definition 6.1. Any WSML vocabulary (see Definition 2.3) is a WSML-Rule vocabulary.

Definition 6.2 defines the set of terms *Term(V)* for a given vocabulary *V*.

Definition 6.2. Any WSML term (see Definition 2.4) is a WSML Rule term.

As usual, the set of ground terms *GroundTerm(V)* is the maximal subset of *Term(V)* which does not contain variables.

Definition 6.3. Given a set of WSML-Rule terms $Term_{Rule}(V)$, an atomic formula in *L*(*V*) is defined by:

- If $r \in V_R$ and $t_1, ..., t_n$ are terms, then $r(t_1, ..., t_n)$ is an atomic formula in *L*(*V*).
- If $\alpha, \beta \in Term_{Rule}(V)$ then α **=** β, and α **!=** β are atomic formulae in *L*(*V*).
- If $\alpha, \beta \in Term_{Rule}(V)$ and $\gamma \in Term(V)$ or γ is of the form { $\gamma_1,...,\gamma_n$ } with $\gamma_1,...,\gamma_n \in Term_{Rule}(V)$, then:
  - α **subConceptOf** γ is an atomic formula in *L*(*V*)
  - α **memberOf** γ is an atomic formula in *L*(*V*)
  - α[β **ofType** γ] is an atomic formula in *L*(*V*)
  - α[β **impliesType** γ] is an atomic formula in *L*(*V*)
  - α[β **hasValue** γ] is an atomic formula in *L*(*V*)

A ground atomic formula is an atomic formula with no variables.

Definition 6.4. Given a WSML-Rule vocabulary *V*, the set of formulae in *L(V)* is recursively defined as follows:

- Given a head-formula $\beta \in Head(V)$ and a body-formula $\alpha \in Body(V)$, β **:-** α is a formula. Here we call α the *body* and β the *head* of the formula. The formula is admissible if (1) α is an admissible body formula, (2) β is an admissible head formula, (3) the safety condition holds and (4) the resulting WSML-Rule knowledge base is *locally stratified*.
- We define the set of admissible head formula *Head(V)* as follows:
  - Any atomic formula α which does not contain the inequality symbol (!=) or the unification operator (=) is in *Head(V)*.
  - Let $\alpha,\beta \in Head(V)$, then α **and** β is in *Head(V)*.
  - α **implies** β is in *Head(V)* .
  - α **impliedBy** β is in *Head(V)*.
  - α **equivalent** β is in *Head(V)*.
- Any admissible head formula in *Head(V)* is a formula in *L(V)*.
- We define the set of admissible body formulae *Body(V)* as follows:
  - Any atomic formula α is in *Body(V)*
  - For $\alpha \in Body(V)$, **naf** α is in *Body(V)*.
  - For $\alpha,\beta \in Body(V)$, α **and** β is in *Body(V)*.
  - For $\alpha,\beta \in Body(V)$, α **or** β is in *Body(V)*.
- Any formula of the form !- α with $\alpha \in Body(V)$ is an admissible formula and is called a *constraint*.

As with the general WSML logical expression syntax, <-, -> and <-> can be seen as synonyms of the keywords

inplies, **impliedBy** and **equivalent**, respectively.

We require each WSML-Rule knowledge base to be *locally stratified*. Appendix A of [Kifer et al., 1995] explains local stratification for a frame-based logical language.

The following are examples of WSML-Rule logical expressions:

- f(?y) memberOf ?z impliedBy ?z memberOf ?x :- naf ?y[a hasValue f(g(?x)), start hasValue _date(2005,6,6,0,0), nr hasValue 10, name hasValue "myName"] and p(?z).

6.2. Differences between WSML-Flight and WSML-Rule

WSML-Rule allows unsafe rules and the use of function symbols in the language.

7. WSML-Full

WSML-Full combines First-Order Logic with nonmonotonic negation in order to provide an expressive language which is able to capture all aspects of Ontology and Web Service modeling. Furthermore, WSML-Full unifies the Description Logic and Logic Programming variants of WSML, namely, WSML-DL and WSML-Rule, in a principled way, under a common syntactic and semantic umbrella.

The goal of WSML-Full is to allow the full syntactic freedom of a First-Order logic and the full syntactic freedom of a Logic Programming language with default negation in a common semantic framework. The challenge for WSML-Full is to find an extension of First-Order Logic which can properly capture default negation. One (obvious) possible extension is Reiter's default logic [Reiter, 1987]. However, the semantics of WSML-Full is at the moment still an open research issue.

Note that both the conceptual and logical expression syntax for WSML-Full completely corresponds with the WSML syntax introduced in Chapter 2. Note also that the WSML-Full logical expression syntax is similar to the logical language specified in WSMO D2 v1.1 [Roman et al., 2004].

7.1. Differences between WSML-DL and WSML-Full

WSML-Full adds full first-order modeling: n-ary predicates, function symbols and chaining variables over predicates. Furthermore, WSML-Full allows non-monotonic negation.

7.2. Differences between WSML-Rule and WSML-Full

WSML-Full adds disjunction, classical negation, multiple model semantics, and the equality operator.

8. WSML Semantics

In the previous chapters we have defined the conceptual and logical expression syntax for different WSML variants. We have mentioned several characteristics of the semantics of different variants, but we have not defined the semantics itself. This chapter specifies the formal semantics for the WSML variants, in this version of the document of WSML-Core, WSML-Flight and WSML-Full. The semantics of WSML-DL and WSML-Full are future work.

At this stage, the semantics of capability descriptions is not entirely clear. Therefore, we only define the semantics of ontology definitions.

In the following we provide first a mapping between the conceptual syntax for ontologies and the logical expression syntax for that part of the conceptual syntax which has a meaning in the logical language. We then provide a semantics for WSML-Core, WSML-Flight and WSML-Rule through mapping to existing logical formalisms. Finally, we will demonstrate that these languages are properly layered.

8.1. Mapping Conceptual Syntax to Logical Expression Syntax

In order to be able to specify the WSML semantics in a concise and understandable way, we first translate the conceptual syntax to the logical expression syntax.

Before we translate the conceptual syntax to the logical expression syntax, we perform the following pre-processing steps:

- Introduce unnumbered anonymous identifiers for missing identifiers.
- Remove all non-functional properties from the conceptual model.
- Replace idlists with single ids for **subRelationOf**.
  E.g., "P **subRelationOf** {Q, R}" is substituted by "P **subRelationOf** Q" and "P **subRelationOf** R".
- Expand all sQNames to full IRIs using the namespace declarations.

Table 8.1 contains the mapping between the WSML conceptual syntax for ontologies and the logical expression syntax through the mapping function τ (*X* and *Y* are meta-variables and are replaced with actual identifiers or variables during the translation itself; p_{new} is a newly introduced predicate). In the table, italic keywords refer to productions in the WSML grammar (see Appendix A) and boldfaced keywords refer to keywords in the WSML language.

WSML Conceptual Syntax	WSML Logical Expression Syntax
τ(*ontology*)	τ(*ontology_element$_1$*) ... τ(*ontology_element$_n$*)
τ(**concept** *id* *superconcept* *attribute$_1$* ... *attribute$_n$*)	τ(*superconcept*, *id*) τ(*attribute$_1$*, *id*) ... τ(*attribute$_n$*, *id*)
τ(**subConceptOf** *idlist*, *X*)	*X* **subConceptOf** *idlist*.
τ(*attribute_id* *attributefeature* **impliesType** *cardinality* *range_idlist*, *X*)	**?x memberOf** *X* **and ?x[*attribute_id* hasValue ?y] implies ?y memberOf** *range_idlist*. τ(*cardinality*, *X*, *attribute_id*) τ(*attributefeature*, *X*, *attribute_id*)
τ(*attribute_id* *attributefeature* **ofType** *cardinality* *range_idlist*, *X*)	**!- ?x memberOf** *X* **and ?x[***attribute_id* **hasValue ?y] and naf ?y memberOf** *range_idlist*. τ(*cardinality*, *X*, *attribute_id*) τ(*attributefeature*, *X*, *attribute_id*)
τ(**transitive**, *X*, *Y*)	**?x memberOf** *X* **and ?y memberOf** *X* **and ?x[***Y* **hasValue ?y] and ?y[***Y* **hasValue ?z] implies ?x[***Y* **hasValue ?z].**
τ(**symmetric**, *X*, *Y*)	**?x memberOf** *X* **and ?y memberOf** *X* **and ?x[***Y* **hasValue ?y] implies ?y[***Y* **hasValue ?x].**
τ(**reflexive**, *X*, *Y*)	**?x memberOf** *X* **implies ?x[***Y* **hasValue ?x].**
τ(**inverseOf**(*att_id*), *X*, *Y*)	**?x memberOf** *X* **and ?x[***Y* **hasValue ?y] implies ?y[***att_id* **hasValue ?x].** **?y memberOf** *X* **and ?x[***att_id* **hasValue ?y] implies ?y[***Y* **hasValue ?x].**
τ(**(***n***)**, *X*, *Y*)	τ(**(***n n***)**, *X*, *Y*)
τ(**(***n m***)**, *X*, *Y*)	τ(**(***n* ***)**, *X*, *Y*) τ(**(**0 *m***)**, *X*, *Y*)
τ(**(***n* ***)**, *X*, *Y*)	p_{new}**(?x) :- ?x memberOf** *X* **and ?x[***Y* **hasValue** $?y_1$, ..., *Y* **hasValue** $?y_n$**] and** $?y_1$ **!=** $?y_2$ **and** $?y_1$ **!=** $?y_3$ **and ... and** $?y_{n-1}$ **!=** $?y_n$. **!- ?x memberOf** *x* **and naf** p_{new}**(?x).**
τ(**(**0 *m***)**, *X*, *Y*)	**!- ?x memberOf** *X* **and ?x[***Y* **hasValue** $?y_1$, ..., *Y* **hasValue** $?y_{m+1}$**] and** $?y_1$ **!=** $?y_2$ **and** $?y_1$ **!=** $?y_3$ **and ... and** $?y_m$ **!=** $?y_{m+1}$.
τ(**relation** *id*/*arity* *superrelation*)	τ(*superrelation*, *id*, *arity*)
τ(**subRelationOf** *id*, *X*, *Y*)	***X*(?x$_1$,...,?x$_Y$) implies *id*(?x$_1$,...,?x$_Y$).**
τ(**instance** *id* *memberof* *attributevalue$_1$* ... *attributevalue$_n$*)	τ(*memberof*, *id*) τ(*attributevalue$_1$*, *id*) τ(*attributevalue$_n$*, *id*)
τ(**memberOf** *idlist*, *X*)	*X* **memberOf** *idlist*.
τ(*att_id* **hasValue** *valuelist*, *X*)	*X***[***att_id* **hasValue** *valuelist***].**
τ(**axiom** *id* *log_expr*)	*log_expr*

Table 8.1: Mapping WSML conceptual syntax to logical expression syntax.

As an example, we translate the following WSML ontology:

```
namespace {_"http://www.example.org/ontologies/example#",
    dc      _"http://purl.org/dc/elements/1.1#",
    foaf _"http://xmlns.com/foaf/0.1/",
    xsd     _"http://www.w3.org/2001/XMLSchema#",
    wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
    loc     _"http://www.wsmo.org/ontologies/location#",
    oo      _"http://example.org/ooMediator#"}

ontology Family
    nfp
        dc#title hasValue "WSML example ontology"
    endnfp

    concept Human subConceptOf { Primate, LegalAgent }
        nonFunctionalProperties
            dc#description hasValue "concept of a human being"
        endNonFunctionalProperties
        hasName ofType foaf#name
```

```
relation ageOfHuman/2 (ofType Human, ofType _integer)
    nfp
        dc#relation hasValue FunctionalDependencyAge
    endnfp

axiom FunctionalDependencyAge
    definedBy
        !- ageOfHuman(?x,?y1) and
        ageOfHuman(?x,?y2) and wsml#numericInequal(?y1,?y2).
```

To the following logical expressions:

```
_"http://www.example.org/ontologies/example#Human"[_"http://www.example.org/ontologies/example#hasName" ofType
  _"http://xmlns.com/foaf/0.1/name"].
_"http://www.example.org/ontologies/example#Human" subConceptOf
  { _"http://www.example.org/ontologies/example#Primate",
  _"http://www.example.org/ontologies/example#LegalAgent" }.

!- naf ?x memberOf _"http://www.example.org/ontologies/example#Human" and _"http://www.example.org/ontologies/example#ageOfHuman"(?x,?y).
!- naf ?y memberOf _integer and _"http://www.example.org/ontologies/example#ageOfHuman"(?x,?y).
!- _"http://www.example.org/ontologies/example#ageOfHuman"(?x,?y1) and
     _"http://www.example.org/ontologies/example#ageOfHuman"(?x,?y2) and _"http://www.wsmo.org/wsml/wsml-syntax#numericInequal"(?y1,?y2).
```

8.2. Preprocessing steps

In order to make the definition of the WSML semantics more straightforward, we define a number of preprocessing steps to be applied to the WSML logical expressions. We identify the following preprocessing steps in order to obtain a suitable set of logical expressions which can be readily mapped to a logical formalism:

Replacing idlists with with multiple statements

Statements involving argument lists of the form *A op* {v_1,...,v_n}, with *op* ∈ {**hasValue**, **ofType**, **impliesType**}, are replaced by multiple statements as such: *A op* v_1, **...**, *A op* v_n.

Statements involving argument lists of the form *A is-a* {c_1,...,c_n}, with *is-a* ∈ {**memberOf**, **subConceptOf**}, are replaced by a conjunction of statements as such: *A op* c_1 **and** ... **and** *A op* c_n.

Reducing composed molecules to single molecules

Composed molecules are split into singular molecules in two steps:

- Molecules of the form *a is-a b*[c_1 op_1 d_1,..., c_n op_n d_n], with *is-a* ∈ {**memberOf**, **subConceptOf**} and op_i ∈ {**hasValue**, **ofType**, **impliesType**} are transformed to: *a is-a b* **and** *a*[c_1 op_n d_1,..., c_n op_n d_n]
- Then, attributes of the form *a*[c_1 op_1 d_1,..., c_n op_n d_n], with op_i ∈ {**hasValue**, **ofType**, **impliesType**}, are translated to: *a*[c_1 op_1 d_1] **and** ... **and** *a*[c_n op_n d_n]

Replacing equivalence with two implications

lexpr **equivalent** *rexpr*. := *lexpr* **implies** *rexpr*. *lexpr* **impliedBy** *rexpr*.

Replacing right implication with left implication.

lexpr **implies** *rexpr*. := *rexpr* **impliedBy** *lexpr*.

Rewriting data term shortcuts

The shortcuts for writing strings, integers and decimals are rewritten to their full form:
"*string*" := _string("*string*")   (unless "*string*" already occurs in the _string datatype wrapper)
integer := _integer("*integer*")
decimal := _decimal("*decimal*")

Rewrite data terms to predicates

Data terms occur as functions in WSML. However, Datalog does not allow the use of function symbols. Thus, we rewrite the datatype wrappers to built-in predicates as such:
Each datatype wrapper with arity *n* has a corresponding built-in predicate with the same name as the datatype wrapper (cf. Appendix C). This built-in predicate always has an arity *n*+1. Each occurrence of a datatype wrapper δ in a statement φ is replaced with a new variable ?x and the datatype predicate corresponding to the wrapper δ is conjoined with the obtained statement φ': (φ' **and** δ(X_1,...,X_1,**?x**)).

Rewrite built-in functions to predicates

Built-in functions are replaced with predicates similar to datatype wrappers. Each of the built-in predicates corresponding to built-in functions mentioned in Appendix C contains one argument which is the result. The occurrence of the function is replaced with a variable and the statement is replaced with the conjunction of that statement and the built-in predicate.

Unfolding sQNames to full IRIs

Finally, all sQNames in the syntax are replaced with full IRIs, according to the rules defined in Section 2.2.

The resulting set of logical expressions does not contain any syntactical shortcuts and can be used directly for the definition of the semantics of the respective WSML variants.

8.3. WSML-Core Semantics

In order to define the semantics of WSML-Core, we first define the notion of a WSML-Core knowledge base in Definition 8.1.

Definition 8.1. We define a WSML-Core knowledge base *KB* as a collection of formulas written in the WSML logical expression language which are the result of application of the translation function τ of Table 8.1 and the preprocessing steps defined in Section 8.2 to a WSML-Core ontology.

We define the semantics of WSML-Core through a mapping to Horn logic using the mapping function π.

Table 8.2 presents the WSML-Core semantics through a direct mapping to function-free Horn logic. In the table, *id#* can be any identifier, *dt#* is a datatype identifier, *X#* can be either a variable or an identifier. Each occurrence of x and each occurrence of y represents a newly introduced variable.

WSML	Horn logic
π(*head* **impliedBy** *body*.)	π(*head*) ← π(*body*)
π(*lexpr* **or** *rexpr*)	π(*lexpr*) ∨ π(*rexpr*)
π(*lexpr* **and** *rexpr*)	π(*lexpr*) ∧ π(*rexpr*)
π(*X1* **memberOf** *id2*)	*id2*(*X1*)
π(*id1* **subConceptOf** *id2*)	*id2*(x) ← *id1*(x)
π(*X1*[*id2* **hasValue** *X2*])	*id2*(*X1*,*X2*)
π(*id1*[*id2* **impliesType** *id3*])	*id3*(y) ← *id1*(x) ∧ *id2*(x,y)
π(*id1*[*id2* **ofType** *dt*])	*dt*(y) ← *id1*(x) ∧ *id2*(x,y)
π($p(X_1,...,X_n)$)	$p(X_1,...,X_n)$

Table 8.2: WSML-Core Semantics

Note that in the translation of typing constraints using **ofType**, we introduce a (built-in) datatype predicate in the head of the rule. However, rule engines typically do not allow built-in predicates in the head of a rule. For implementation in database/rule systems, this formula can be rewritten to the following constraint: ← *id1*(x) ∧ *id2*(x,y) ∧ not *dt*(y), with 'not' being default negation. Similar for the use of **impliesType** with a datatype as range.

Each occurrence of an unnumbered anonymous ID is replaced with a new globally unique identifier. All occurrences of the same numbered anonymous ID in one formula are replaced with the same new globally unique identifier.

Application of the usual Lloyd-Topor transformations [Lloyd and Topor, 1984] yield actual Datalog rules. In particular, the following transformations are iteratively applied until no transformation is applicable:

- Rules of the form $A_1 \wedge ... \wedge A_n \leftarrow B$ are split in *n* different rules:
  - $A_1 \leftarrow B$
  - ...
  - $A_n \leftarrow B$
- Rules of the form $A_1 \leftarrow A_2 \leftarrow B$ are transformed to:
  - $A_1 \leftarrow A_2 \wedge B$
- Rules of the form $A \leftarrow B_1 \vee ... \vee B_n$ are split into *n* different rules:
  - $A \leftarrow B_1$
  - ...
  - $A \leftarrow B_n$

Definition 8.2 (Satisfiability in WSML-Core) We say a WSML-Core knowledge base *KB* is satisfiable iff π(KB_A) is satisfiable under first-order semantics.

Definition 8.3 (Entailment in WSML-Core) We say a WSML-Core knowledge base KB_A entails a WSML-Core knowledge base KB_B, written as: $KB_A \models_{WSML} KB_B$ iff $\pi(KB_A) \models \pi(KB_B)$, where |= is the classical entailment relation.

8.4. WSML-Flight Semantics

In order to define the semantics of WSML-Flight, we first define the notion of a WSML-Flight knowledge base in Definition 8.4.

Definition 8.4. We define a WSML-Flight knowledge base *KB* as a collection of formulas written in the WSML logical expression language which are the result of application of the translation function τ of Table 8.1 and the preprocessing steps defined in Section 8.2 to a WSML-Flight ontology.

We define the semantics of WSML-Flight through a mapping to the Datalog fragment of F-Logic [Kifer et al., 1995] (extended with inequality and stratified default negation in the body of the rule) using the mapping function π.

Concepts, instances and attributes are interpreted as objects in F-Logic. We need a number of auxiliary rules in order to ensure the correct interpretation of the translated F-Logic statements (with *not* denoting default negation and a rule with an empty head denoting an integrity constraint):

The semantics of method signatures is captured through an integrity constraint on method signatures:

$\leftarrow x[y \Rightarrow z] \wedge w{:}x \wedge w[y \rightarrow v] \wedge \text{not } v{:}z$

The semantics of 'impliesType' is captured through an auxiliary predicate:

$v{:}z \leftarrow _impliestype(x,y,z) \wedge w{:}x \wedge w[y \rightarrow v]$

Now follows the semantics of WSML-Flight in Table 8.3. In the table, *X#* stands for either a variable or an identifier; '=' is the unification operator and '!=' is the built-in inequality symbol.

WSML	F-Logic
π(**!-** *body*.)	← π(*body*)
π(*head* **:-** *body*.)	π(*head*) ← π(*body*)
π(*lexpr* **impliedBy** *rexpr*.)	π(*lexpr*) ← π(*rexpr*)
π(*lexpr* **or** *rexpr*)	π(*lexpr*) ∨ π(*rexpr*)
π(*lexpr* **and** *rexpr*)	π(*lexpr*) ∧ π(*rexpr*)
π(*X1* **memberOf** *X2*)	*X1*:*X2*
π(*X1* **subConceptOf** *X2*)	*X1*::*X2*
π(*X1*[*X2* **hasValue** *X3*])	*X1*[*X2* → *X3*]
π(*X1*[*X2* **ofType** *X3*])	*X1*[*X2* ⇒ *X3*]
π(*X1*[*X2* **impliesType** *X3*])	_impliestype(*X1*,*X2*,*X3*)
π($p(X_1,...,X_n)$)	$p(X_1,...,X_n)$
π($X_1 = X_2$))	$X_1 = X_2$)
π(X_1 != X_2)))	X_1 != X_2)

Table 8.3: Semantics of WSML-Flight

Rules with empty heads are integrity constraints. The first row in the table produces integrity constraints from the WSML logical expressions. Furthermore, there exists the integrity constraint which axiomatizes the semantics of **ofType**. All integrity constraints are part of the set of integrity constraints *C*.

Each occurrence of an unnumbered anonymous ID is replaced with a new globally unique identifier. All occurrences of the same numbered anonymous ID in one formula are replaced with the same new globally unique identifier.

Application of the usual Lloyd-Topor transformations [Lloyd and Topor, 1984] yield actual Datalog rules. In particular, the following transformations are iteratively applied until no transformation is applicable:

- Rules of the form $A_1 \wedge ... \wedge A_n \leftarrow B$ are split in *n* different rules:
    - $A_1 \leftarrow B$
    - ...
    - $A_n \leftarrow B$
- Rules of the form $A_1 \leftarrow A_2 \leftarrow B$ are transformed to:
    - $A_1 \leftarrow A_2 \wedge B$
- Rules of the form $A \leftarrow B_1 \vee ... \vee B_n$ are split into *n* different rules:
    - $A \leftarrow B_1$
    - ...
    - $A \leftarrow B_n$

We base the semantics of WSML-Flight on the perfect model semantics by Przymusinski [Przymusinski, 1989], which defines the semantics for locally stratified logic programs. Przymusinski shows that every stratified program

has a unique perfect model. WSML-Flight only allows locally stratified negation.

Definition 8.5 (Satisfiability in WSML-Flight) Say, we have a WSML-Flight knowledge base *KB* and a set of constraints *C*. *KB* is satisfiable iff $\pi(KB)$ has a perfect model M_{KB} which does not violate any of the constraints in *C*. We say an integrity constraint is *violated* if some ground instantiation of the body of the constraint is true in the model M_{KB}.

We define the semantics of WSML-Flight with respect to the entailment of ground formulas. We say a formula is ground if it does not contain any variables.

Definition 8.6 (Entailment in WSML-Flight) We say a satisfiable WSML-Flight knowledge base *KB* entails a WSML-Flight ground formula *F* iff $M_{KB} \models \pi(F)$, where M_{KB} is the perfect model of *KB*.

8.5. WSML-Rule Semantics

The semantics of WSML-Rule is defined similar to WSML-Flight. The only difference is that the semantics of WSML-Rule is not defined through a mapping to Datalog, but through a mapping to full Logic Programming (i.e., with function symbols and allowing unsafe rules) with inequality and (locally) stratified negation. However, the mapping looks exactly the same as Table 8.3. The only difference is that the meta-variables *X#* can also stand for a constructed term.

In order to define the semantics of WSML-Rule, we first define the notion of a WSML-Rule knowledge base in Definition 8.4.

Definition 8.7. We define a WSML-Rule knowledge base *KB* as a collection of formulas written in the WSML logical expression language which are the result of application of the translation function τ of Table 8.1 and the preprocessing steps defined in Section 8.2 to a WSML-Rule ontology.

We define the semantics of WSML-Rule through a mapping to the Horn fragment of F-Logic [Kifer et al., 1995] (extended with inequality and locally stratified default negation in the body of the rule) using the mapping function π.

Concepts, instances and attributes are interpreted as objects in F-Logic. We need a number of auxiliary rules in order to ensure the correct interpretation of the translated F-Logic statements (with *not* denoting default negation and a rule with an empty head denoting an integrity constraint):

The semantics of method signatures is captured through an integrity constraint on method signatures:

$\leftarrow x[y \Rightarrow z] \land w{:}x \land w[y \rightarrow v] \land \text{not } v{:}z$

The semantics of 'impliesType' is captured through an auxiliary predicate:

$v{:}z \leftarrow _impliestype(x,y,z) \land w{:}x \land w[y \rightarrow v]$

Now follows the semantics of WSML-Rule in Table 8.3. In the table, *X#* stands for either a variable, an identifier, or a constructed term; '=' is the unification operator and '!=' is the built-in inequality symbol.

WSML	F-Logic
π(**!-** *body*.)	← π(*body*)
π(*head* **:-** *body*.)	π(*head*) ← π(*body*)
π(*lexpr* **impliedBy** *rexpr*.)	π(*lexpr*) ← π(*rexpr*)
π(*lexpr* **or** *rexpr*)	π(*lexpr*) ∨ π(*rexpr*)
π(*lexpr* **and** *rexpr*)	π(*lexpr*) ∧ π(*rexpr*)
π(*X1* **memberOf** *X2*)	*X1*:*X2*
π(*X1* **subConceptOf** *X2*)	*X1*::*X2*
π(*X1*[*X2* **hasValue** *X3*])	*X1*[*X2* → *X3*]
π(*X1*[*X2* **ofType** *X3*])	*X1*[*X2* ⇒ *X3*]
π(*X1*[*X2* **impliesType** *X3*])	_impliestype(*X1*,*X2*,*X3*)
π($p(X_1,...,X_n)$)	$p(X_1,...,X_n)$
π($X_1 = X_2$))	$X_1 = X_2$)
π(X_1 != X_2)))	X_1 != X_2)

Table 8.4: Semantics of WSML-Rule

Rules with empty heads are integrity constraints (in the database sense). The first row in the table produces integrity constraints from the WSML logical expressions. Furthermore, there exists the integrity constraint which axiomatizes the semantics of **ofType**. All integrity constraints are part of the set of integrity constraints *C*.

Each occurrence of an unnumbered anonymous ID is replaced with a new globally unique identifier. All occurrences of the same numbered anonymous ID in one formula are replaced with the same new globally unique identifier.

Application of the usual Lloyd-Topor transformations [Lloyd and Topor, 1984] yield actual Datalog rules (note that the syntactical restrictions on the WSML-Rule logical expression syntax prevent the use of disjunction in the head of any rule). In particular, the following transformations are iteratively applied until no transformation is applicable:

- Rules of the form $A_1 \wedge ... \wedge A_n \leftarrow B$ are split in *n* different rules:
  - $A_1 \leftarrow B$
  - ...
  - $A_n \leftarrow B$
- Rules of the form $A_1 \leftarrow_{LT} A_2 \leftarrow B$ are transformed to:
  - $A_1 \leftarrow A_2 \wedge B$
- Rules of the form $A \leftarrow B_1 \vee ... \vee B_n$ are split into *n* different rules:
  - $A \leftarrow B_1$
  - ...
  - $A \leftarrow B_n$

We base the semantics of WSML-Rule on the perfect model semantics by Przymusinski [Przymusinski, 1989], which defines the semantics for (locally) stratified logic programs. Przymusinski shows that every locally stratified program has a unique perfect model. WSML-Rule only allows locally stratified negation.

Definition 8.8 (Satisfiability in WSML-Rule) Say, we have a WSML-Rule knowledge base *KB* and a set of constraints *C*. *KB* is satisfiable iff π(*KB*) has a perfect model M_{KB} which does not violate any of the constraints in *C*. We say an integrity constraint is *violated* if some ground instantiation of the constraint is true in the model M_{KB}.

We define the semantics of WSML-Rule with respect to the entailment of ground formulas. We say a formula is ground if it does not contain any variables.

Definition 8.9 (Entailment in WSML-Rule) We say a satisfiable WSML-Rule knowledge base *KB* entails a WSML-Rule ground formula *F* iff M_{KB} |= π(*F*), where M_{KB} is the perfect model of *KB*.

PART III: THE WSML EXCHANGE SYNTAXES

In the previous part we have described the WSML family of languages in terms of their human-readable syntax. This syntax might not be suitable for exchange between automated agents. Therefore, we present three exchange syntaxes for WSML in order to enable automated interoperation.

The three exchange syntaxes for WSML are:

XML syntax:
: A syntax specifically tailored for machine processability, instead of human-readability; it is easily parsable by standard XML parsers, but is quite unreadable for humans. This syntax is defined in Chapter 9.

RDF syntax
: An alternate exchange syntax for WSML is WSML/RDF. WSML/RDF can be used to leverage the currently existing RDF tools, such a triple stores, and to syntactically combine WSML/RDF descriptions with other RDF descriptions. WSML/RDF is defined in Chapter 10.

Mapping to OWL
: A bidirectional mapping between (a subset of) OWL and WSML is given in Chapter 11.

9 XML Syntax for WSML

In this chapter, we explain the XML syntax for WSML. The XML syntax for WSML is based on the human-readable syntax for WSML presented in Chapter 2. The XML syntax for WSML captures all WSML variants. The user can specify the variant of a WSML/XML document through the 'variant' attribute of the <wsml> root element.

The complete XML Schema, including documentation, for the WSML/XML syntax can be found in Appendix B. Table 9.1 provides the transformations of the conceptual syntax, Table 9.2 the transformation of logical expressions, while in Table 9.3 a simple mapping example is given.

The basic namespace for WSML/XML is http://www.wsmo.org/wsml/wsml-syntax#. This is the namespace for all elements in WSML.

Before beginning the transformation process the following pre-processing steps need to be performed:

- sQNames need to be resolved to full IRIs, i.e., there will not be any namespace definitions in the XML syntax.
- in the conceptual syntax universal truth, universal falsehood and unnumbered anonymous IDs are resolved to: *http://www.wsmo.org/wsml/wsml-syntax#true*, *http://www.wsmo.org/wsml/wsml-syntax#false*, and *http://www.wsmo.org/wsml/wsml-syntax#anonymousID*, respectively. Numbered anonymous IDs are resolved as such: _#1 to *http://www.wsmo.org/wsml/wsml-syntax#anonymousID1*, _#2 to *http://www.wsmo.org/wsml/wsml-syntax#anonymousID2*, ..., _#*n* to *http://www.wsmo.org/wsml/wsml-syntax#anonymousID*n.
- Datatype identifiers are resolved to full IRIs by replacing the leading underscore '_' with the WSML namespace. For example, the datatype identifier '_date' is resolved to *http://www.wsmo.org/wsml/wsml-syntax#integer*.

T(...) denotes a function which is recursively called to transform the first parameter (a fragment of WSML syntax) to the XML syntax.

In Table 9.1, all WSML keywords are marked bold. *A,B,C* stand for identifiers, *D* stands for a datatype identifier, DV_i stands for an integer value, DV_d stands for a decimal, and DV_s stands for a string data value. *k,m,n* are integer numbers. Productions in the grammar are underlined and linked to the production rules in Appendix A. Note that in the table we use the familiar sQName macro mechanism (see also Section 2.1.2) to abbreviate the IRIs of resources. The prefix 'xsd' stands for 'http://www.w3.org/2001/XMLSchema#'.

WSML syntax	XML Tree	Remarks
T (**wsmlVariant** *A* definition$_1$ ... definition$_n$)	<**wsml** xmlns="http://www.wsmo.org/wsml/wsml-syntax#" variant="*A*"> T(definition$_1$) ... T(definition$_n$) </**wsml**>	definitions are Ontology, Goal, WebService and Mediators
T (**namespace** { *N*, P_1 N_1, ..., P_n N_n })		Because sQnames were resolved to full IRIs during pre-processing, there is no translation to RDF necessary for namespace definitions
T (**ontology** *A* header$_1$ ... header$_n$ ontology_element$_1$ ... ontology_element$_n$)	<**ontology** name="*A*"> T(header$_1$) ... T(header$_n$) T(ontology_element$_1$) ... T(ontology_element$_n$) </**ontology**>	An ontology_element represents all possible content of an ontology definition, i.e., concepts, relations, instances, ...
T (**concept** *C* **subConceptOf** {B_1,...,B_n} nfp attribute$_1$ ... attribute$_n$)	<**concept** name="*C*"> T(nfp) <**superConcept**>B_1<**superConcept**> ... <**superConcept**>B_n<**superConcept**> T(attribute$_1$) ... T(attribute$_n$) </**concept**>	
T (*A* attributefeature$_1$ ... attributefeature$_n$ **ofType** cardinality *C* nfp)	<**attribute** name="*A*" type="constraining"> <**range**>*C*</**range**> T(attributefeature$_1$) ... T(attributefeature$_n$) T(cardinality) T(nfp) </**attribute>**	
T (*A* attributefeature$_1$ ... attributefeature$_n$ **impliesType** cardinality *C* nfp)	<**attribute** name="*A*" type="inferring"> <**range**>*C*</**range**> T(attributefeature$_1$) ... T(attributefeature$_n$) T(cardinality) T(nfp) </**attribute>**	
T (**transitive**)	<**transitive**/>	
T (**symmetric**)	<**symmetric**/>	
T (**reflexive**)	<**reflexive**/>	
T (**inverseOf**(*A*))	<**inverseOf**>*A*</**inverseOf**>	
T ((*minCard maxCard*))	<**minCardinality**>*minCard*</**minCardinality**> <**maxCardinality**>*maxCard*</**maxCardinality**>	

Table 9.1: Mapping the WSML conceptual syntax to the WSML/XML syntax

WSML syntax	XML Tree	Remarks
T ((*card*))	<**minCardinality**>*card*</**minCardinality**> <**maxCardinality**>*card*</**maxCardinality**>	
T (**instance** *A* **memberOf** C_1,...,C_n nfp attributevalue$_1$ ... attributevalue$_n$)	<**instance** name="*A*"> <**memberOf**>C_1</**memberOf**> ... <**memberOf**>C_n</**memberOf**> T(nfp) T(attributevalue$_1$) ... T(attributevalue$_n$) </**instance**>	
T (*A* **hasValue** {value$_1$, ..., value$_n$})	<**attributeValue** name="*A*"> T(value$_1$ ... T(value$_n$ </**attributeValue**>	A value has always a datatype. There are four built-in datatypes: IRI, string, integer, decimal. In addition any arbitrary datatype can be defined by use of datatype wrappers. The next four transformations show how to handle these five cases.
T (*A*)	<**value** type="xsd#anyURI"> *A* </**value**>	
T (DV_s)	<**value** type="xsd#string"> DV_s </**value**>	
T (DV_i)	<**value** type="xsd#integer"> DV_i </**value**>	
T (DV_d)	<**value** type="xsd#decimal"> DV_d </**value**>	
T (*DV*)	<**value** type="http://www.example.org/datatype#any"> <**argument**>*DV.arg*$_1$</**argument**> ... <**argument**>*DV.arg*$_n$</**argument**> </**value**>	A datatype wrapper is defined by the type and a number of arguments that define the value, e.g., _date(2005,12,12) for December 12, 2005 (and thus the value of type date has three arguments). In case there is just on argument, the element <argument> is optional and the value can be given as above for IRI, string, integer, decimal.
T (**relation** *A* / *n* (paramtype$_1$,...,paramtype$_m$) **subRelationOf** {C_1,...,C_k} nfp)	<**relation** name="*A*" arity="*n*"> <**parameters**> T(paramtype$_1$>) ... T(paramtype$_n$) </**parameters**> <**superRelation**>C_1</**superRelation**> ... <**superRelation**>C_k</**superRelation**> T(nfp) </**relation**>	Note that the parameters of a relation are ordered and thus the order of the parameters elements in the XML representation important.
T (**ofType** {C_1,...,C_n})	<**parameter** type="constraining"> <**range**>C_1</**range**> ... <**range**>C_n</**range**> </**parameter**>	
T (**impliesType** {C_1,...,C_n})	<**parameter** type="inferring"> <**range**>C_1</**range**> ... <**range**>C_n</**range**> </**parameter**>	

WSML syntax	XML Tree	Remarks
T (**relationInstance** *A B* (value$_1$,...,value$_n$) nfp)	<**relationInstance** name="*A*"> <**memberOf**>*B*</**memberOf**> T(value$_1$) ... T(value$_n$) T(nfp) </**relationInstance**>	
T ({value$_1$,...,value$_n$})	<**parameterValue**> T(value$_1$) ... T(value$_n$) </**parameterValue**>	Note that the parameters of a relationInstance are ordered and thus the order of the value elements is important.
T (**axiom** *A* axiomdefinition)	<**axiom** name="*A*"> T(axiomdefinition) </**axiom**>	
T (nfp **definedBy** log_expr)	T(nfp) <**definedBy**> T(log_expr) </**definedBy**>	The mapping of logical expressions is defined in Table 9.2.
T (**goal** *A* header$_1$ ... header$_n$ capability interface$_1$ ... interface$_n$)	<**goal** name="*A*"> T(header$_1$) ... T(header$_n$) T(capability) T(interface$_1$) ... T(interface$_n$) </**goal**>	
T (**ooMediator** *A* nfp importsontology source target use_service)	<**ooMediator** name="*A*"> T(nfp) T(importsontology) T(source) T(target) T(use_service) </**ooMediator**>	
T (**ggMediator** *A* header$_1$ ... header$_n$ source target use_service)	<**ggMediator** name="*A*"> T(header$_1$) ... T(header$_n$) T(source) T(target) T(use_service) </**ggMediator**>	
T (**wgMediator** *A* header$_1$ ... header$_n$ source target use_service)	<**wgMediator** name="*A*"> T(header$_1$) ... T(header$_n$) source target use_service </**wgMediator**>	
T (**wwMediator** *A* header$_1$ ... header$_n$ source target use_service)	<**wwMediator** name="*A*"> T(header$_1$) ... T(header$_n$) T(source) T(target) T(use_service) </**wwMediator**>	

WSML syntax	XML Tree	Remarks
T (**source** { A_1,...,A_n })	<**source**>A_1</**source**> ... <**source**>A_n</**source**>	
T (**target** *A*)	<**target**>*A*</**target**>	
T (**usesService** *A*)	<**usesService**>*A*</**usesService**>	
T (**webService** *A* $header_1$ ... $header_n$ capability $interface_1$ ... $interface_n$)	<**webService** name="*A*"> T($header_1$) ... T($header_n$) T(capability) T($interface_1$) ... T($interface_n$) </**webService**>	
T (**capability** *C* $header_1$ ... $header_n$ sharedvardef $pre_post_ass_or_eff_1$ ... $pre_post_ass_or_eff_n$)	<**capability** name="*C*"> T($header_1$) ... T($header_n$) T(sharedvardef) T($pre_post_ass_or_eff_1$) ... T($pre_post_ass_or_eff_n$) </**capability**>	pre_post_ass_or_eff unites the axiom definitions for precondition, assumption, postcondition and effect.
T (**sharedVariables** {$?v_1$, ..., $?v_n$})	<**sharedVariables**> <**variable** name="$?v_1$"/> ... <**variable** name="$?v_n$"/> </**sharedVariables**>	
T (**precondition** *B* axiomdefinition)	<**precondition** name="*B*"> T(axiomdefinition) </**precondition**>	
T (**assumption** *B* axiomdefinition)	<**assumption** name="*B*"> T(axiomdefinition) </**assumption**>	
T (**postcondition** *B* axiomdefinition)	<**postcondition** name="*B*"> T(axiomdefinition) </**postcondition**>	
T (**effect** *B* axiomdefinition)	<**effect** name="*B*"> T(axiomdefinition) </**effect**>	
T (**interface** *A* $header_1$ ... $header_n$ choreography orchestration)	<**interface** name="*A*"> T($header_1$) ... T($header_n$) T(choreography) T(orchestration) </**interface**>	
T (**choreography** *C*)	<**choreography**>*C*</**choreography**>	

WSML syntax	XML Tree	Remarks
T (**orchestration** *A*)	<**orchestration**>*A*</**orchestration**>	
T (**nonFunctionalProperties** $\underline{\text{attributevalue}}_1$ ... $\underline{\text{attributevalue}}_n$ **endNonFunctionalProperties**)	<**nonFunctionalProperties**> T($\underline{\text{attributevalue}}_1$) ... T($\underline{\text{attributevalue}}_n$) </**nonFunctionalProperties**>	
T (**nfp** $\underline{\text{attributevalue}}_1$ ... $\underline{\text{attributevalue}}_n$ **endnfp**)	<**nonFunctionalProperties**> T($\underline{\text{attributevalue}}_1$) ... T($\underline{\text{attributevalue}}_n$) </**nonFunctionalProperties**>	
T (**importsOntology** { A_1,...,A_n})	<**importsOntology**>A_1</**importsOntology**> ... <**importsOntology**>A_n</**importsOntology**>	
T (**usesMediator** {B_1,...,B_n})	<**usesMediator**>B_1</**usesMediator**> ... <**usesMediator**>B_n</**usesMediator**>	

The logical expression syntax is explained in Table 9.2. In the table, *A* stands for identifiers and $T_1,...,T_n$ stand for terms. *V* stands for a variable.

WSML syntax	XML Tree	Remarks
T (*A*(term$_1$, ..., term$_n$))	<**term** name="*A*"> T (term$_1$, arg) ... T (term$_n$, arg) </**term**>	
T (*A*(term$_1$, ..., term$_n$) , *name*)	<***name*** name="*A*"> T (term$_1$, arg) ... T (term$_n$, arg) </***name***>	
T (*V*)	<**term** name="*C*"> </**term**>	
T (*V* , *name*)	<***name*** name="*V*"> </***name***>	
T (*A*(term$_1$, ..., term$_n$))	<**atom** name="*A*"> T(term$_1$, arg) ... T(term$_n$, arg) </**atom**>	
T (*T$_1$*[attr_relation$_1$, ..., attr_relation$_n$] **subConceptOf** *T$_2$*)	<**molecule**> T(*T$_1$*) <**isa** type="subConceptOf"> T(*T$_2$*) </**isa**> T(attr_relation$_1$) ... T(attr_relation$_n$) </**molecule**>	
T (*T$_1$*[attr_relation$_1$, ..., attr_relation$_n$] **memberOf** *T$_2$*)	<**molecule**> T(*T$_1$*) <**isa** type="memberOf"> T(*T$_2$*) </**isa**> T(attr_relation$_1$) ... T(attr_relation$_n$) </**molecule**>	
T (*T* [attr_relation$_1$, ..., attr_relation$_n$])	<**molecule**> T(*T*) T(attr_relation$_1$) ... T(attr_relation$_n$) </**molecule**>	
T (*T$_0$* **ofType** {*T$_1$*, ..., *T$_n$*})	<**attributeDefinition** type="constraining"> T(*T$_0$*, name) T(*T$_1$*, type) ... T(*T$_n$*, type) </**attributeDefinition**>	
T (*T$_0$* **impliesType** {*T$_1$*, ..., *T$_n$*})	<**attributeDefinition** type="inferring"> T(*T$_0$*, name) T(*T$_1$*, type) ... T(*T$_n$*, type) </**attributeDefinition**>	
T (*T$_0$* **hasValue** {*T$_1$*, ..., *T$_n$*})	<**attributeValue**> T(*T$_0$*, name) T(*T$_1$*, value) ... T(*T$_n$*, value)	

Table 9.2: Mapping the WSML logical expression syntax to the WSML/XML syntax

WSML syntax	XML Tree	Remarks
	</**attributeValue**>	
T (expr$_1$ **and** expr$_2$)	<**and**> T(expr$_1$) T(expr$_2$) </**and**>	
T (expr$_1$ **or** expr$_2$)	<**or**> T(expr$_1$) T(expr$_2$) </**or**>	
T (**neg** expr)	<**neg**> T(expr) </**neg**>	
T (**naf** expr)	<**naf**> T(expr) </**naf**>	
T (expr$_1$ **implies** expr$_2$)	<**implies**> T(expr$_1$) T(expr$_2$) </**implies**>	
T (expr$_1$ **impliedBy** expr$_2$)	<**impliedBy**> T(expr$_1$) T(expr$_2$) </**impliedBy**>	
T (expr$_1$ **equivalent** expr$_2$)	<**equivalent**> T(expr$_1$) T(expr$_2$) </**equivalent**>	
T (**forall** $?v_1, ..., ?v_n$ (expr))	<**forall**> <**var**>$?v_1$</**var**> ... <**var**>$?v_n$</**var**> T(expr) </**forall**>	
T (**exists** $?v_1, ..., ?v_n$ (expr))	<**exists**> <**var**>$?v_1$</**var**> ... <**var**>$?v_n$</**var**> T(expr) </**exists**>	
T (**!-** expr)	<**constraint**> T(expr) </**constraint**>	
T (expr$_1$ **:-** expr$_2$)	<**impliedByLP**> T(expr$_1$) T(expr$_2$) </**impliedByLP**>	

Table 9.3 provides a simple translation example.

WSML syntax

```
wsmlVariant
 _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace {_"http://www.example.org/ex1#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
  ex _"http://www.example.org/ex2#"}

ontology _"http://www.example.org/ex1"
  nonFunctionalProperties
    dc#title hasValue "WSML to RDF"
    dc#date hasValue _date(2005,12,12)
  endNonFunctionalProperties

  importsOntology _"http://www.example.net/ex2"

  concept Woman subConceptOf
        {ex#Human, ex#LivingBeing}
    name ofType _string
    ancestorOf transitive impliesType ex#Human
    age ofType (1) _integer

  axiom GenderConstraint
    definedBy
        !- ?x memberOf ex#Man and
          ?x memberOf Woman.

  instance Mary memberOf Woman
    name hasValue "Mary Jones"
    age hasValue 23

  relation childOf/2
    (ofType ex#Human, impliesType ex#Parent)


webService ws
  capability itineraryInfo
  interface
   {_"http://example.org/i1",_"http://example.org/i2"}
```

XML Tree

```
<wsml xmlns="http://www.wsmo.org/wsml/wsml-syntax#"
      variant="http://www.wsmo.org/wsml/wsml-syntax/wsml-flight">
  <ontology name="http://www.example.org/ex1">
    <nonFunctionalProperties>
      <attributeValue name="http://purl.org/dc/elements/1.1#title">
        <value type="http://www.wsmo.org/wsml/wsml-syntax#string">
          WSML to RDF
        </value>
      </attributeValue>
      <attributeValue name="http://purl.org/dc/elements/1.1#date">
        <value type="http://www.wsmo.org/wsml/wsml-syntax#date">
          <argument>2005</argument>
          <argument>12</argument>
          <argument>12</argument>
        </value>
      </attributeValue>
    </nonFunctionalProperties>
    <importsOntology>http://www.example.org/ex2</importsOntology>
    <concept name="http://www.example.org/ex1#Woman">
      <superConcept>
        http://www.example.org/ex2#Human
      <superConcept>
      <superConcept>
        http://www.example.org/ex2#LivingBeing
      <superConcept>
      <attribute name="http://www.example.org/ex1#name" type="constraining">
        <range>http://www.wsmo.org/wsml/wsml-syntax#string</range>
      </attribute>
      <attribute name="http://www.example.org/ex1#ancestor" type="inferring">
        <range>http://www.example.org/ex2#Human</range>
        <transitive/>
      </attribute>
      <attribute name="http://www.example.org/ex1#age" type="inferring">
        <range>http://www.wsmo.org/wsml/wsml-syntax#integer</range>
        <minCardinality>1</minCardinality>
        <maxCardinality>1</maxCardinality>
      </attribute>
    </concept>
    <axiom name="http://www.example.org/ex1#GenderConstraint">
      <definedBy>
        <constraint>
          <and>
            <molecule>
              <term name="?x"/>
              <isa type="memberOf">
                <term name="http://www.example.org/ex1#Woman"/>
              </isa>
            </molecule>
            <molecule>
              <term name="?x"/>
              <isa type="memberOf">
                <term name="http://www.example.org/ex2#Man"/>
              </isa>
            </molecule>
          </and>
        </constraint>
      </definedBy>
    </axiom>
    <instance name="http://www.example.org/ex1#Mary">
      <memberOf>http://www.example.org/ex1#Woman</memberOf>
      <attributeValue name="http://www.example.org/ex1#name">
        <value type="http://www.wsmo.org/wsml/wsml-syntax#string">
          Mary Jones
        </value>
      </attributeValue>
      <attributeValue name="http://www.example.org/ex1#age">
        <value type="http://www.wsmo.org/wsml/wsml-syntax#integer">
          23
        </value>
      </attributeValue>
    </instance>
    <relation name="http://www.example.org/ex1#childOf" arity="2">
      <parameters>
        <parameter type="constraining">
          <range>http://www.example.org/ex2#Human</range>
        </parameter>
        <parameter type="inferring">
          <range>http://www.example.org/ex2#Parent</range>
        </parameter>
      </parameters>
    </relation>
  </ontology>
  <webService name="http://www.example.org/ex1#ws">
    <capability>http://www.example.org/ex1#itineraryInfo</capability>
    <interface>http://example.org/i1</interface>
    <interface>http://example.org/i2</interface>
  </webService>
</wsml>
```

10. RDF Syntax for WSML

In this chapter an RDF syntax for WSML is introduced. The vocabulary used is an extension of the RDF Schema vocabulary defined in [Brickley & Guha, 2004]. The extension consists of WSML language components, as given in Table 10.1.

WSML keyword		**WSML keyword**	
wsml#variant	Used as *predicate* to indicate the WSML variant applied.	wsml#goal	Used to define a WSML goal.
wsml#ontology	Used to define a WSML ontology.	wsml#ooMediator	Used to defined a WSML ooMediator
wsml#hasConcept	Used to type a WSML concept and to bind it to an ontology.	wsml#ggMediator	Used to defined a WSML ggMediator
wsml#attribute	Used to define a WSML attribute	wsml#wgMediator	Used to defined a WSML wgMediator
wsml#ofType	Used as *predicate* to define constraining attributes and parameters.	wsml#wwMediator	Used to defined a WSML wwMediator
wsml#hasAttribute	Used as *predicate* to bind an attribute to a concept.	wsml#webService	Used to defined a WSML webService
wsml#transitiveAttribute	Used to indicate the transitivity of an attribute.	wsml#useInterface	Used as *predicate* to bind an interface to a Web service.
wsml#symmetricAttribute	Used to indicate the symmetry of an attribute.	wsml#useCapability	Used as *predicate* to bind a capability to a Web service or goal.
wsml#reflexiveAttribute	Used to indicate the reflexivity of an attribute.	wsml#sharedVariables	Used as *predicate* to bind a shared variable to a capability.
wsml#inverseOf	Used to indicate the inverse relationship of two attributes.	wsml#precondition	Used to type a precondition and to bind it to a capability.
wsml#minCardinality	Used as *predicate* to defined the minimal cardinality of an attribute.	wsml#assumption	Used to type an assumption and to bind it to a capability.
wsml#maxCardinality	Used as *predicate* to defined the maximal cardinality of an attribute.	wsml#postcondition	Used to type a postcondition and to bind it to a capability.
wsml#hasInstance	Used to type an instance and to bind it to a concept.	wsml#effect	Used to type an effect and to bind it to a capability.
wsml#hasRelation	Used to type a relation and to bind it to an ontology.	wsml#choreography	Used to type a choreography and to bind it to an interface.
wsml#arity	Used to define the arity of a WSML relation.	wsml#orchestration	Used to type a orchestration and to bind it to an interface.
wsml#param	Used to type parameters of WSML relations.	wsml#nfp	Used to type non-functional properties.
wsml#subRelationOf	Used as *predicate* to define sub-relations.	wsml#importsOntology	Used to define the import of an external ontology.
wsml#hasRelationInstance	Used to type a relation instance and to bind it to an ontology.	wsml#usesMediator	Used to define the use of a mediator.
wsml#hasAxiom	Used to type an axiom and to bind it to an ontology.		

Table 10.1: WSML vocabulary extending RDFS for WSML triple syntax

The remainder of this chapter presents a mapping between the human-readable syntax of WSML and the RDF/WSML-triple syntax for all WSML variants, where the knowledge is encoded in *<subject><predicate><object>*-triples. The big advantage of having such a syntax and reusing the RDFS-vocabulary as far as possible, is the fact that there are many existing RDF(S)-based tools available. These tools are optimized to handle triples and are thus able to understand parts of our specification and in a more

general sense we can guarantee inter-operability with those. In RDF all triples are global, while in the conceptual syntax of WSML it is possible to append specific entities to higher-level entities, e.g., concepts to ontologies or attribute values to instances.

In order to maximize the reuse of RDFS vocabulary, several Dublin Core properties are translated to corresponding RDFS properties. Namely, dc#title is mapped to rdfs#label, dc#description is mapped to rdfs#comment, and dc#relation is mapped to rdfs#seeAlso.

Table 10.2 defines the mapping function T from WSML entities to RDF triples. Logical expressions are not completely translated to RDF. instead, they are translated to the WSML/XML syntax and are specified as literals of type rdf#XMLLiteral. The transformation creates a RDF-graph based on above introduced RDF-triples. As definitions, i.e., top-level entities like ontologies, Web services, goals and mediators are disjoint constructs, their graphs are not inter-related. In other words the transformation defines one graph per definition. Note that in the table we use the familiar sQName macro mechanism (see also Section 2.1.2) to abbreviate IRIs. The prefix 'wsml' stands for 'http://www.wsmo.org/wsml/wsml-syntax#', 'rdf' stands for 'http://www.w3.org/1999/02/22-rdf-syntax-ns#', 'rdfs' stands for 'http://www.w3.org/2000/01/rdf-schema#', 'dc' stands for 'http://purl.org/dc/elements/1.1#', and 'xsd' stands for 'http://www.w3.org/2001/XMLSchema#'.

In Table 10.2, *A,B,C,Z* stand for identifiers, *D* stands for a datatype identifier, DV_i stands for an integer value, DV_d stands for a decimal, and DV_s stands for a string data value. *k,m,n* are integer numbers.

The basic namespace for WSML/RDF is http://www.wsmo.org/wsml/wsml-syntax#. This is the namespace for all elements in WSML.

Before beginning the transformation process the following pre-processing steps need to be performed:

- In case multiple top-level specifications (i.e., ontology, goal, web service, mediator) occur in the same document, the document must be split in multiple WSML documents. Each WSML document is then translated to a separate RDF graph.
- sQNames need to be resolved to full IRIs, i.e., there will not be any namespace definitions in the RDF syntax.
- in the conceptual syntax universal truth, universal falsehood and unnumbered anonymous IDs are resolved to: *http://www.wsmo.org/wsml/wsml-syntax#true*, *http://www.wsmo.org/wsml/wsml-syntax#false*, and *http://www.wsmo.org/wsml/wsml-syntax#anonymousID*, respectively. Numbered anonymous IDs are resolved as such: _#1 to *http://www.wsmo.org/wsml/wsml-syntax#anonymousID1*, _#2 to *http://www.wsmo.org/wsml/wsml-syntax#anonymousID2*, ..., _#*n* to *http://www.wsmo.org/wsml/wsml-syntax#anonymousID*n.
- Datatype identifiers are resolved to full IRIs by replacing the leading underscore '_' with the WSML namespace. For example, the datatype identifier '_date' is resolved to *http://www.wsmo.org/wsml/wsml-syntax#integer*.

A simple example of a full translation from the human-readable syntax to the RDF triple notation is given in Table 10.3.

WSML syntax	RDF Triples	Remarks
T (**wsmlVariant** *A* definition$_1$ ... definition$_n$)	T(definition$_1$, *A*) ... T(definition$_n$, *A*)	definitions are Ontology, Goal, WebService and Mediators
namespace { *N*, P_1 N_1 ... P_n N_n }		Because sQnames were resolved to full IRIs during pre-processing, there is no translation to RDF necessary for namespace definitions
T (**ontology** *A* header$_1$ ... header$_n$ ontology_element$_1$ ... ontology_element$_n$, *Z*)	*A* rdf#type wsml#ontology *A* wsml#variant *Z* T(header$_1$, *A*) ... T(header$_n$, *A*) T(ontology_element$_1$, *A*) ... T(ontology_element$_n$, *A*)	An ontology_element represents possible content of an ontology definition, i.e., concepts, relations, instances, ...
T (**concept** *A* **subConceptOf** {B_1,...,B_n} nfp attribute$_1$ ... attribute$_n$, *Z*)	*Z* wsml#hasConcept *A* T(nfp, *A*) *A* rdfs#subClassOf B_1 ... *A* rdfs#subClassOf B_n T(attribute$_1$, *A*) ... T(attribute$_n$, *A*)	
T (*A* attributefeature$_1$ ... attributefeature$_n$ **ofType** cardinality *C* nfp , *Z*)	_:X wsml#attribute *A* T(attributefeature$_1$, _:X) ... T(attributefeature$_n$, _:X) _:X wsml#ofType *C* T(cardinality, _:X) T(nfp, _:X) *Z* wsml#hasAttribute _:X	The blank identifier_:X denotes a helper node to bind the attribute *A* to a defined owner: attributes are locally defined!
T (*A* attributefeature$_1$ ... attributefeature$_n$ **impliesType** cardinality *C* nfp , *Z*)	_:X wsml#attribute *A* T(attributefeature$_1$, _:X) ... T(attributefeature$_n$, _:X) _:X rdfs#range *C* T(cardinality, _:X) T(nfp, _:X) *Z* wsml#hasAttribute _:X	
T (**transitive** , *Z*)	*Z* rdf#type wsml#transitiveAttribute	
T (**symmetric** , *Z*)	*Z* rdf#type wsml#symmetricAttribute	
T (**reflexive** , *Z*)	*Z* rdf#type wsml#reflexiveAttribute	
T (**inverseOf**(*A*) , *Z*)	*Z* wsml#inverseOf *A*	
T ((*m n*) , *Z*)	*Z* wsml#minCardinality *m* *Z* wsml#maxCardinality *n*	
T ((*card*) , *Z*)	*Z* wsml#minCardinality *card* *Z* wsml#maxCardinality *card*	

Table 10.2: Mapping to the WSML/RDF syntax

WSML syntax	RDF Triples	Remarks
T (**instance** *A* **memberOf** C_1,...,C_n nfp attributevalue$_1$ ... attributevalue$_n$, *Z*)	*Z* wsml#hasInstance *A* *A* rdf#type C_1 ... *A* rdf#type C_n T(nfp, *A*) T(attributevalue$_1$, *A*) ... T(attributevalue$_n$, *A*)	It is not required to associate an instance with an identifier. In that case the identifier *J* is replaced by a blank node identifier, e.g _:X. The same counts for all entities that do not require an ID: instance, relationInstance, but also goal, mediators, webService and capabilty and interface definitions.
T (dc#title **hasValue** value , *Z*)	*Z* rdfs#label T(value)	
T (dc#description **hasValue** value , *Z*)	*Z* rdfs#comment T(value)	
T (dc#relation **hasValue** value , *Z*)	*Z* rdfs#seeAlso T(value)	
T (*A* **hasValue** value , *Z*)	*Z A* T(value)	
T (DV_s)	DV_s^^xsd#string	String are already enclosed with double quotes in WSML; these do not have to be added for the RDF literals.
T (DV_i)	"DV_i"^^xsd#integer	
T (DV_d)	"DV_d"^^xsd#decimal	
T ($D(a_1,...,a_n)$)	$T_{serializer}(D(a_1,...,a_n))$^^$T_{datatypes}(D)$	$T_{serializer}$ serializes the WSML representation of a data value to a string representation which can be readily used in RDF literals. This function is not yet defined. $T_{datatypes}$ maps WSML datatypes to XML Schema datatypes, according to Table C.1 in Appendix C.
T (*A*)	*A*	IRIs are directly used in RDF.
T (**relation** *A* / *n* (B_1,...B_m) **subRelationOf** {C_1,...,C_k} nfp , *Z*)	*Z* wsml#hasRelation *A* *A* wsml#arity "*n*"^^xsd#integer *A* wsml#param _:X _:X rdf#type rdf#List _:X rdf#first _:X$_1$ T(B_1, _:X$_1$) _:X rdf#rest _:1 _:1 rdf#type rdf#List _:1 rdf#first _:X$_2$ ... _:m rdf#type rdf#List _:m rdf#first _:X$_n$ T(B_m, _:X$_n$) *R* wsml#subRelationOf C_1 ... *R* wsml#subRelationOf C_k T(nfp, *A*)	The parameters of a relation are unnamed and thus ordered. The ordering in RDF is provided by use of the rdf#List.
T (**ofType** *C* , *Z*)	*Z* wsml#ofType *C*	

WSML syntax	RDF Triples	Remarks
T (**impliesType** *C* , *Z*)	*Z* rdfs#range *C*	
T (**relationInstance** *A B* (valuelist) nfp , *Z*)	*Z* wsml#hasRelationInstance *A* *A* rdf#type *B* T(valuelist, *A*) T(nfp, *A*)	
T ($value_1$,...,$value_n$, *Z*)	*Z* wsml#param _:X _:X rdf#type rdf#List _:X rdf#first T($value_1$) _:X rdf#rest _:1 _:1 rdf#type rdf#List _:1 rdf#first T($value_2$) _:1 rdf#rest _:2 ... _:m rdf#type rdf#List _:m rdf#first T($value_n$)	The arguments of a relationinstance are unnamed and thus ordered. The ordering in RDF is provided by use of the rdf#List.
T (**axiom** *A* axiomdefinition , *Z*)	*Z* wsml#hasAxiom *A* T(axiomdefinition, *A*)	
T (nfp **definedBy** log_expr)	T(nfp, *A*) *A* rdfs#isDefinedBy "T_{XML}(log_expr)"^^rdf#XMLLiteral	log_expr denotes a logical expression. The logical expressions are translated to literals of type rdf#XMLLiteral using the mapping function defined in Table 9.2
T (**goal** *A* $header_1$ ... $header_n$ capability $interface_1$ ... $interface_n$, *Z*)	*A* wsml#variant *Z* *A* rdf#type wsml#goal T($header_1$, *A*) ... T($header_n$, *A*) T(capability, *A*) T($interface_1$, *A*) ... T($interface_n$, *A*)	
T (**ooMediator** *A* nfp importsontology source target use_service , *Z*)	*A* wsml#variant *Z* *A* rdf#type wsml#ooMediator T(nfp, *A*) T(importsontology, *A*) T(source, *A*) T(target, *A*) T(use_service, *A*)	
T (**ggMediator** *A* nfp importsontology source target use_service , *Z*)	*A* wsml#variant *Z* *A* rdf#type wsml#ggMediator T(nfp, *A*) T(importsontology, *A*) T(source, *A*) T(target, *A*) T(use_service, *A*)	
T (**wgMediator** *A* nfp importsontology source target use_service , *Z*)	*A* wsml#variant *Z* *A* rdf#type wsml#wgMediator T(nfp, *A*) T(importsontology, *A*) T(source, *A*) T(target, *A*) T(use_service, *A*)	
T (**wwMediator** *A* nfp importsontology	*A* wsml#variant *Z* *A* rdf#type wsml#wMediator T(nfp, *A*) T(importsontology, *A*)	

WSML syntax	RDF Triples	Remarks
source target use_service , *Z*)	T(source, *A*) T(target, *A*) T(use_service, *A*)	
T (**source** { A_1,...,A_n } , *Z*)	*Z* wsml#source A_1 ... *Z* wsml#source A_n	
T (**target** *A* , *Z*)	*Z* wsml#target *A*	
T (**usesService** *A* , *Z*)	*Z* wsml#usesService *A*	
T (**webService** *A* $header_1$ ... $header_n$ capability $interface_1$ ... $interface_n$, *Z*)	*A* wsml#variant *Z* *A* rdf#type wsml#webService T($header_1$, *A*) ... T($header_n$, *A*) T(capability, *A*) T($interface_1$, *A*) ... T($interface_n$, *A*)	
T (**capability** *C* $header_1$ ... $header_n$ sharedvardef $pre_post_ass_or_eff_1$ ... $pre_post_ass_or_eff_n$, *Z*)	*Z* wsml#useCapabilty *C* T($header_1$, *C*) ... T($header_n$, *C*) T(sharedvardef, *C*) T($pre_post_ass_or_eff_1$, *C*) ... T($pre_post_ass_or_eff_n$, *C*)	pre_post_ass_or_eff reunites the axiom definitions for precondition, assumption, postcondition and effect.
T (**sharedVariables** {?v_1, ..., ?v_n} , *Z*)	*Z* wsml#sharedVariables ?v_1 ... *Z* wsml#sharedVariables ?v_n	Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.
T (**precondition** *B* axiomdefinition , *Z*)	*Z* wsml#precondition *B* T(axiomdefinition, *B*)	
T (**assumption** *B* axiomdefinition , *Z*)	*Z* wsml#assumption *B* T(axiomdefinition, *B*)	
T (**postcondition** *B* axiomdefinition , *Z*)	*Z* wsml#postcondition *B* T(axiomdefinition, *B*)	
T (**effect** *B* axiomdefinition , *Z*)	*Z* wsml#effect *B* T(axiomdefinition, *B*)	
T (**interface** *A* $header_1$ ... $header_n$ choreography orchestration , *Z*)	*Z* wsml#useInterface *A* T($header_1$, *A*) ... T($header_n$, *A*) T(choreography, *A*) T(orchestration, *A*)	

WSML syntax	RDF Triples	Remarks
T (**choreography** *C* , *Z*)	*Z* wsml#choreography *C*	
T (**orchestration** *A* , *Z*)	*Z* wsml#orchestration *A*	
T (**nonFunctionalProperties** attributevalue$_1$ ... attributevalue$_n$ **endNonFunctionalProperties** , *Z*)	*Z* wsml#nfp _:P$_1$ T(attributevalue$_1$, _:P$_1$) ... *Z* wsml#nfp _:P$_n$ T(attributevalue$_n$, _:P$_n$)	
T (**nfp** attributevalue$_1$ ... attributevalue$_n$ **endnfp** , *Z*)	*Z* wsml#nfp _:P$_1$ T(attributevalue$_1$, _:P$_1$) ... *Z* wsml#nfp _:P$_n$ T(attributevalue$_n$, _:P$_n$)	
T (**importsOntology** { A_1,...,A_n} , *Z*)	*Z* wsml#importsOntology A_1 ... *Z* wsml#importsOntology A_n	
T (**usesMediator** {B_1,...,B_n} , *Z*)	*Z* wsml#usesMediator B_1 ... *Z* wsml#usesMediator B_n	

Table 10.3 provides a simple translation example.

WSML syntax	RDF Triples
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight" **namespace** {_"http://www.example.org/ex1#", dc _"http://purl.org/dc/elements/1.1#", wsml _"http://www.wsmo.org/wsml/wsml-syntax#", ex _"http://www.example.org/ex2#"} **ontology** _"http://www.example.org/ex1" **nonFunctionalProperties** dc#title **hasValue** "WSML to RDF" dc#date **hasValue** _date(2005,12,12) **endNonFunctionalProperties** **importsOntology** _"http://www.example.net/ex2" **concept** Woman **subConceptOf** {ex#Human, ex#LivingBeing} name **ofType** _string ancestorOf **transitive impliesType** ex#Human age **ofType** (1) _integer **axiom** GenderConstraint **definedBy** **!-** ?x **memberOf** ex#Man **and** ?x **memberOf** Woman. **instance** Mary **memberOf** Woman name **hasValue** "Mary Jones" age **hasValue** 23 **relation** childOf/2 (**ofType** ex#Human, **impliesType** ex#Parent) **webService** ws **capability** itineraryInfo **interface** {_"http://example.org/i1",_"http://example.org/i2"}	The first RDF graph for the ontology: http://www.example.org/ex1 rdf#type wsml#ontology http://www.example.org/ex1 wsml#variant http://www.wsmo.org/wsml/wsml-syntax/wsml-flight http://www.example.org/ex1 wsml#nfp _:P1 _:P1 http://purl.org/dc/elements/1.1#title "WSML to RDF"^^xsd#string http://www.example.org/ex1 wsml#nfp _:P2 _:P2 http://purl.org/dc/elements/1.1#date "2005-02-04"^^xsd#date http://www.example.org/ex1 wsml#importsOntology http://www.example.org/ex2 http://www.example.org/ex1 wsml#hasConcept http://www.example.org/ex1#Woman http://www.example.org/ex1#Woman rdfs#subClassOf http://www.example.org/ex2#Human http://www.example.org/ex1#Woman rdfs#subClassOf http://www.example.org/ex2#LivingBeing http://www.example.org/ex1#Woman wsml#hasAttribute _:X _:X wsml#ofType xsd#string _:X wsml#attribute http://www.example.org/ex1#name http://www.example.org/ex1#Woman wsml#hasAttribute _:Y _:Y rdfs#range http://www.example.org/ex2#Human _:Y rdf#type wsml#transitiveAttribute _:Y wsml#attribute http://www.example.org/ex1#ancesterOf http://www.example.org/ex1#Woman wsml#hasAttribute _:Z _:Z wsml#ofType xsd#integer _:Z wsml#minCardinality 1 _:Z wsml#maxCardinality 1 _:Z wsml#attribute http://www.example.org/ex1#age http://www.example.org/ex1 wsml#hasAxiom http://www.example.org/ex1#GenderConstraint http://www.example.org/ex1#WomanDef rdfs#isDefinedBy "<constraint>... </constraint>"^^rdf#XMLLiteral http://www.example.org/ex1 wsml#hasInstance http://www.example.org/ex1#Mary http://www.example.org/ex1#Mary rdf#type http://www.example.org/ex1#Woman http://www.example.org/ex1#Mary http://www.example.org/ex1#name "Mary Jones"^^xsd#string http://www.example.org/ex1#Mary http://www.example.org/ex1#age "23"^^xsd#integer http://www.example.org/ex1 wsml#hasRelation http://www.example.org/ex1#childOf http://www.example.org/ex1#childOf wsml#arity "2"xsd#integer http://www.example.org/ex1#childOf wsml#hasParameter _:A _:A wsml#ofType http://www.example.org/ex2#Human http://www.example.org/ex1#childOf wsml#hasParameter _:B _:B rdfs#range http://www.example.org/ex2#Parent The second RDF graph for the web service description: http://example.org/ws rdf#type wsml#webService http://example.org/ws wsml#variant http://www.wsmo.org/wsml/wsml-syntax/wsml-flight http://example.org/ws wsml#useCapability http://example.org/itineraryInfo http://example.org/ws wsml#useInterface http://example.org/i1 http://example.org/ws wsml#useInterface http://example.net/i2

Table 10.3: A Mapping Example

11 Mapping to OWL

The mapping to OWL presented here is applicable to ontologies and logical expressions only; note that logical expressions might occur in ontologies, as well as goal and web service capability descriptions. For a mapping of non-ontology constructs except logical expressions the RDF Syntax for WSML has to be used. Furthermore this version of the deliverable contains only a mapping of WSML-Core to OWL. Once WSML-DL has been specified, a mapping of WSML-DL to OWL will be defined. Other WSML variants will not be mapped directly to OWL, since their semantics are not compatible. If a mapping is desired first such an ontology has to be reduced to either WSML-DL or WSML-Core.

11.1. Mapping WSML-Core to OWL DL

In this section we define a mapping of WSML-Core to the OWL DL abstract syntax [Patel-Schneider et al., 2004].

In order to simplify the translation we perform the following pre-processing steps:

- Replace missing ontology identifier by the locator of the file containing the specification.
- Replace missing identifiers with unnumbered anonymous identifiers (i.e., for concepts, relations, instances, relation instances and axioms)
- Replace all unnumbered anonymous identifier by **http://www.wsmo.org/wsml/wsml-syntax#anonymousID**
- Replace idlists with single ids (in the case of **ofType**, **impliesType**, **hasValue**, **subConceptOf** and **subRelationOf**).
  E.g., "hasAnchestor **hasValue** {Peter, Paul}" is substituted by "hasAnchestor **hasValue** Peter" and "hasAnchestor **hasValue** Paul".
- All sQNames in the syntax are replaced with full IRIs, according to the rules defined in Section 2.2
- Specifically within logical expression the following pre-processing steps are applied:
  - Replacing right implication with left implication (*lexpr* **implies** *rexpr*. := *rexpr* **impliedBy** *lexpr*).
  - Rewriting data term shortcuts ("string" := _string("string"); integer := _integer("integer"); decimal := _decimal("decimal")).
  - Rewriting all WSML datatype constructor according to Table C.1 to their corresponding XML schema datatype, e.g., **_string** to **http://www.w3.org/2001/XMLSchema#string**

Table 11.1 contains the mapping between the WSML Core and OWL DL abstract syntax through the mapping function τ. In the table, underlined words refer to productions rules in the WSML grammar (see Appendix A) and boldfaced words refer to keywords in the WSML language. *X* and *Y* are meta-variables and are replaced with actual identifiers or variables during the translation itself. Note that in the table we use the familiar sQName macro mechanism (see also Section 2.1.2) to abbreviate IRIs. The prefix 'wsml' stands for 'http://www.wsmo.org/wsml/wsml-syntax#', 'rdf' stands for 'http://www.w3.org/1999/02/22-rdf-syntax-ns#', 'rdfs' stands for 'http://www.w3.org/2000/01/rdf-schema#', 'dc' stands for 'http://purl.org/dc/elements/1.1#', 'xsd' stands for 'http://www.w3.org/2001/XMLSchema#', and 'owl' stands for 'http://www.w3.org/2002/07/owl#'.

WSML-Core conceptual syntax	OWL DL Abstract syntax	Remarks
Mapping for ontologies		
τ(**ontology** *A* header$_1$ ... header$_n$ ontology_element$_1$ ... ontology_element$_n$)	Ontology(*id* τ(header$_1$) ... τ(header$_n$) τ(ontology_element$_1$) ... τ(ontology_element$_n$))	
τ(**nonFunctionalProperties** id_1 **hasValue** $value_1$ ... id_n **hasValue** $value_n$ **endNonFunctionalProperties**)	annotation(id_1 τ($value_1$)) ... annotation(id_n τ($value_n$))	For non-functional properties on the ontology level "Annotation" instead of "annotation" has to be written.
τ(**importsOntoloy** *idlist*)	Annotation(owl#import id_1) ... Annotation(owl#import id_n)	An *idlist* can consist of n full IRIs (this remark holds for all idlists in this table.
τ(**usesMediator** *idlist*)	Annotation(wsml#usesMediator id_1) ... Annotation(wsml#usesMediator id_n)	OWL does not have the concept of a mediator. Therefore, the translation uses the wsml#usesMediator annotation.
Mapping for concepts		
τ(**concept** *id* *superconcept* *nfp* *att_id*$_1$ *att_type*$_1$ *range_id*$_1$ ... *att_id*$_n$ *att_type*$_n$ *range_id*$_n$)	Class(id partial τ(nfp) τ(*superconcept*) restriction (*att_id*$_1$ allValuesFrom *range_id*$_1$)... restriction (*att_id*$_n$ allValuesFrom *range_id*$_n$)) [ObjectProperty|DatatypeProperty] (*att_id*$_1$) ... [ObjectProperty|DatatypeProperty] (*att_id*$_n$)	"DatatypeProperty" is chosen in case *range_id*$_i$ refers to a datatype; otherwise, "ObjectProperty" is used.
τ(**subConceptOf** *idlist*)	id_1 ... id_n	
Mapping for relations		
τ(**relation** *id* *arity* *paramtyping* *superrelation* *nfp*)	[ObjectProperty|DatatypeProperty] (*id* τ(*nfp*) τ(*superrelation*) τ(*paramtyping*))	"DatatypeProperty" is chosen in case *range_id* refers to a datatype identifier; otherwise the relation is mapped to "ObjectProperty" (*range_id* refers to the second type of *paramtyping*).
τ(**subRelationOf** *idlist*)	super(id_1) .. super(id_n)	
τ((*att_type* *domain_id*, *att_type* *range_id*))	domain(*domain_id*) range(*range_id*)	
Mapping for instances		
τ(**instance** *id* *memberof* *nfp* *att_id*$_1$ **hasValue** $value_1$ ... *att_id*$_n$ **hasValue** $value_n$)	Individual(*id* τ(*nfp*) τ(*memberof*) value (*att_id*$_1$ τ($value_1$)) ... value (*att_id*$_n$ τ($value_n$)))	
τ(**memberOf** *idlist*)	type(id_1)...type(id_n)	
τ(*datatype_id*(x_1,...,x_n))	$τ_{serializer}$(*datatype_id*(x_1,...,x_n))^^$τ_{datatypes}$(*datatype_id*)	$τ_{serializer}$ serializes the WSML representation of a data value to a string representation which can be readily used in OWL. This function is not yet defined. $τ_{datatypes}$ maps WSML datatypes to XML Schema datatypes, according to Table C.1 in Appendix C.
τ(*id*)	*id*	In WSML and IRI is enclosed by _" and ", which are omitted in OWL abstract syntax.
τ(**relationInstance** *relation_id* *memberof_id*($value_1$, $value_2$) *nfp*)	Individual($value_1$ value (*memberof_id* τ($value_2$)))	If the relation instance has an identifier it will be omitted. The NFPs are disregarded as their semantics cannot be captured by OWL.
Mapping for axioms		
τ(**axiom** *id* *nfp* *log_expr*)	τ(*log_expr*)	
τ(*conjunction*)	intersectionOf(τ($molecule_1$), ..., τ($molecule_n$))	

Table 11.1: Mapping between WSML-Core and OWL DL abstract syntax

WSML-Core conceptual syntax	OWL DL Abstract syntax	Remarks
τ(*id*[*att_id₁ att_type₁ range_id₁* ... *att_idₙ att_typeₙ range_idₙ*]*superconcept*)	Class(*id* partial τ(*superconcept*) restriction (*att_id₁* allValuesFrom *range_id₁*)... restriction (*att_idₙ* allValuesFrom *range_idₙ*))	
τ(*id*[*att_id₁* **hasValue** *value₁* ... *att_idₙ* **hasValue** *valueₙ*]*memberof*)	Individual(*id* τ(*memberof*) value (*att_id₁* τ(*value₁*)) ... value (*att_idₙ* τ(*valueₙ*)))	
τ(*?x*[*att_id* **hasValue** *?x*] **impliedBy** *?x*[*att_id* **hasValue** *?y*] **and** *?y*[*att_id* **hasValue** *?z*])	ObjectProperty(*att_id* Transitive)	
τ(*?x*[*att_id* **hasValue** *?y*] **impliedBy** *?y*[*att_id* **hasValue** *?x*])	ObjectProperty(*att_id* Symmetric)	
τ(*?x*[*att_id* **hasValue** *?y*] **impliedBy** *?y*[*att_id2* **hasValue** *?x*])	ObjectProperty(*att_id* InverseOf(*att_id2*))	
τ(*?x* **memberOf** *concept_id2* **impliedBy** *?x* **memberOf** *concept_id*)	Class (*concept_id* partial *concept_id2*)	
τ(*?x* **memberOf** *concept_id* **equivalent** *?x* **memberOf** *concept_id₁* **and** ... **and** *?x* **memberOf** *concept_idₙ*)	Class (*concept_id* complete *concept_id₁* ... *concept_idₙ*)	
τ(*att_id*(*?x*,*?y*) **impliedBy** *att_id2*(*?x*,*?y*))	SubProperty (*att_id att_id2*)	
τ(*?x* **memberOf** *concept_id* **impliedBy** *att_id*(*?x*,*?y*))	ObjectProperty(*att_id* domain(*concept_id*))	
τ(*?y* **memberOf** *concept_id* **impliedBy** *att_id*(*?x*,*?y*))	ObjectProperty(*att_id* range(*att_id2*))	
τ()		If τ is applied for a non-occurring production no translation has to be made

11.2. Mapping OWL DL to WSML-Core

Table 11.2 shows the mapping between OWL DL (abstract syntax) and WSML-Core. The table shows for each construct supported by OWL DL (and being semantically within WSML-Core) the corresponding WSML-Core syntax in terms logical expressions necessary to capture the construct.

The mapping is done by a recursive translation function τ. The symbol X denotes a meta variable that has to be substituted with the actual variable occurring during the translation. Note that only those ontologies within the expressivity of WSML Core, can be translated. If for an OWL DL ontology a mapping can not be found by applying τ the ontology is not within the expressivity of WSML Core.

In order to translate class axioms, the translation function τ takes and WSML logical expression from the translation of the left-hand side, τ_{lhs}, of a subclass relation (or partial/complete class descriptions, respectively) as the first argument, the right hand side of a subclass relationship as second argument, and a variable as third argument. This variable is used for relating classes through properties from the allValuesFrom and someValuesFrom restrictions. Whenever we pass such a value restriction during the translation, a new variable has to be introduced, i.e., x_{new} stands for a freshly introduced variable in every translation step.

In the table we apply the following convention for the Identifiers:

- *A* refer to named classes
- *C* and *D* refer to named class descriptions
- *R* refers to a property
- *V* refers to a Value (either data value or an Individual)
- *I* refers to an Individual

OWL DL Abstract syntax	WSML-Core syntax	Remarks
Class Axioms		
Class(A partial C_1 ... C_n)	**concept** A **nonFunctionalProperties** dc#relation **hasValue** *AxiomOfA* **endNonFunctionalProperties** **axiom** *AxiomOfA* **definedBy** τ(?x_{new} **memberOf** A, C_1, ?x_{new}). ... τ(?x_{new} **memberOf** A, C_n, ?x_{new}).	In case C_i is a named class the axiom definition for this i can be omitted and only conceptual syntax can be used by including C_i in the list of super concepts of A: concept A **subConceptOf** {C_i}
Class(A complete C_1 ... C_n)	**concept** *A* **nonFunctionalProperties** dc#relation **hasValue** *AxiomOfA* **endNonFunctionalProperties** **axiom** *AxiomOfA* **definedBy** τ(?x_{new} **memberOf** A, C_1, ?x_{new}). ... τ(?x_{new} memberOf A, C_n, ?x_{new}). τ(τ_{lhs}(C_1, ?x_{new}), A, ?x_{new}). τ(τ_{lhs}(C_n, ?x_{new}), A, ?x_{new}).	τ_{lhs} stands for a transformation function for left hand side descriptions.
EquivalentClasses(C_1 ... C_n)	**axiom** _# **definedBy** τ(τ_{lhs}(C_i, ?x_{new}), C_j, ?x_{new}).	Conjunctively for all $i \neq j$
SubClassOf(C_1 C_2)	**axiom** _# **definedBy** τ(τ_{lhs}(C_1, ?x_{new}), C_2, ?x_{new})	
Mapping of left hand side descriptions		
τ_{lhs}(A,X)	X **memberOf** A	
τ_{lhs}(intersectionOf(C_1 ... C_n), X)	τ_{lhs}(C_1, X) **and** ... **and** τ_{lhs}(C_n, X)	
τ_{lhs}(unionOf(C_1 ... C_n), X)	τ_{lhs}(C_1, X) **or** ... **or** τ_{lhs}(C_n, X)	
τ_{lhs}(restriction(R someValuesFrom C), X)	X[R **hasValue** ?x_{new}] **and** τ_{lhs}(C, x_{new})	
τ_{lhs}(restriction(R minCardinality(1)), X)	X[R **hasValue** x_{new}]	
Mapping of right hand side descriptions		
τ(*WSMLExpr*, intersectionOf(C_1 ... C_n), X)	τ(*WSMLExpr*, C_1, X) **and** ... **and** τ(*WSMLExpr*, C_n, X)	
τ(X **memberOf** A_1, A_2, X)	A_1 **subConceptOf** A_2	
τ(*WSMLExpr*, A, X)	*WSMLExpr* **implies** X **memberOf** *A*	
τ(*WSMLExpr*, restriction(R allValuesFrom C), X)	τ(*WSMLExpr* **and** X[R **hasValue** ?x_{new}], C, ?x_{new})	
Property Axioms		
ObjectProperty(R [super(R_1) ... super(R_n)] [domain(C_1) ... domain(C_n)] [range(D_1) ... range(D_n)]	**relation** R/2 **subRelationOf** {R_1, ..., R_n} **nonFunctionalProperties** dc#relation **hasValue** *AxiomOfR* **endNonFunctionalProperties** **axiom** *AxiomOfR* **definedBy** τ(?x[R **hasValue** ?y], C_1, ?x) **and** ... **and** τ(?x[R **hasValue** ?y], C_n, ?x).	All Specifications of the ObjectProperty except the identifier are optional. If both range and domain are given and consist only of named classes their translation can be abbreviated in the conceptual model: relation R(**impliesType** {C_1, ..., C_n} **impliesType** {D_1, ..., D_n})

OWL DL Abstract syntax	WSML-Core syntax	Remarks
[inverseOf(*R'*)] [Symmetric] [Transitive])	τ(?x[*R* **hasValue** ?y], D_1, ?x) **and** ... **and** τ(?x[*R* **hasValue** ?y], D_n, ?x). ?x[*R* **hasValue** ?y] **implies** ?y[*R'* **hasValue** ?x] **and** ?x[*R'* **hasValue** ?y] **implies** ?y[*R* **hasValue** ?x]. ?x[*R* **hasValue** ?y] **implies** ?y[*R* **hasValue** ?x]. ?x[*R* **hasValue** ?y] **and** ?y[*R* **hasValue** ?z]. **implies** ?x[*R* **hasValue** ?z].	
SubProperty(R_1 R_2)	**relation** R_1/2 **subRelationOf** R_2	
EquivalentProperties(R_1 ... R_n)	?x[R_i **hasValue** ?y] **implies** ?x[R_j **hasValue** ?y]	Conjunctively for all i ≠ j
Individuals		
Individual (*I* [type (C_1) ... type(C_n)] [value (R_1 V_1) ... value (R_n V_n)])	**instance** *I* **memberOf** {C_1,..., C_n} R_1 **hasValue** V_1 .... R_n **hasValue** V_n	All Specifications of the Individual except the identifier are optional.

We will now illustrate the translation described in table 11.2 by an example:

Class(Chardonay partial Wine restriction(hasColor value White))

1. translation step: τ(Class(Chardonay partial Wine restriction(hasColor value White)))
   ⇒
   concept Chardonay **subConceptOf** Wine
   nonFunctionalProperties
   dc#relation **hasValue** AxiomChardonay
   endNonFunctionalProperties

   axiom AxiomChardonay
   definedBy
   τ(?x_2 **memberOf** Chardonay, restriction(hasColor value (White)), ?x_2).

2. translation step: τ(?x_2 **memberOf** Chardonay, restriction(hasColor value (White)), ?x_2)
   ⇒
   concept Chardonay **subConceptOf** Wine
   nonFunctionalProperties
   dc#relation **hasValue** AxiomChardonay
   endNonFunctionalProperties

   axiom AxiomChardonay
   definedBy
   ?x_2 **memberOf** Chardonay **implies** ?x_2[hasFlavour **hasValue** White].

PART IV: FINALE

12. Implementation Efforts

WSMO4J (http://wsmo4j.sourceforge.net/, which will provide a data model in Java for WSML and will also provide (de-)serializers for the different WSML syntaxes. WSMO4J can be extended to connect with the specific reasoners to be used for WSML.

The **WSML validator** (http://dev1.deri.at:8080/wsml/) currently provides validation services for the basic syntax defined in D2v1.0 [Roman et al., 2004]. We expect the validator to be extended to handle the different WSML variants under development in this deliverable. However, we expect that the functionality of the WSML validator will eventually be subsumed by WSMO4J, although the validator itself will still be available as a Web Service.

WSMX provides the reference implementation for WSMO. WSMX makes use of pluggable reasoning services. WSMX is committed to using the WSML language developed in this deliverable.

Implementation of **reasoning services** for the different WSML variants is currently under investigation in WSML deliverable D16.2 [de Bruijn, 2004]. Future versions of that deliverable will provide reasoning implementations for the different WSML variants, based on existing reasoning implementations.

Converters will be developed to convert between the different syntaxes of WSML, namely, the human-readable syntax, the XML syntax and the RDF syntax. Furthermore, an importer/exporter for OWL will be created. [JB: here, we need to decide what syntaxes of OWL we accept and which syntaxes we export. Considerations are the abstract syntax, the XML syntax and the RDF syntax. Perhaps we can use a third-party tool to convert between OWL syntaxes and just select the most suitable one for out purposes?]

Appendix A. Human-Readable Syntax

This appendix presents the complete grammar for the WSML language. The language use write this grammar is a variant of Extended Backus Nauer Form which can be interpreted by the SableCC compiler compiler [SableCC].

We present one WSML grammar for all WSML variants. The restrictions that each variants poses on the use of the syntax are described in the respective chapters in PART II of this deliverable.

A.1. BNF-Style Grammar

In this section we show the entire WSML grammar. The grammar is specified using a dialect of Extended BNF which can be used directly in the SableCC compiler compiler [SableCC]. Terminals are quoted, non-terminals are underlined and refer to the tokens and productions. Alternatives are separated using vertical bars '|', and are labeled, where the label is enclosed in curly brackets '{' '}'; optional elements are appended with a question mark '?'; elements that may occur zero or more times are appended with an asterisk '*'; elements that may occur one or more times are appended with a plus '+'.

The first part of the grammar file provides *HELPERS* which are used to write *TOKENS*. Broadly, a language has a collection of tokens (words, or the vocabulary) and rules, or *PRODUCTIONS*, for generating sentences using these tokens (grammar). A grammar describes an entire language using a finite set of productions of tokens or other productions; however this finite set of rules can easily allow an infinite range of valid sentences of the language they describe. Note, helpers cannot directly be used in productions. A last word concerning the *IGNORED TOKENS*: ignored tokens are ignored during the parsing process and are not taken into consideration when building the abstract syntax tree.

Helpers

```
all                    = [ 0x0 .. 0xffff ]
escape_char            = '\'
basechar               = [ 0x0041 .. 0x005A ] | [ 0x0061 .. 0x007A ]
ideographic            = [ 0x4E00 .. 0x9FA5 ] | 0x3007 | [ 0x3021 .. 0x3029 ]
letter                 = basechar | ideographic
digit                  = [ 0x0030 .. 0x0039 ]
combiningchar          = [ 0x0300 .. 0x0345 ] | [ 0x0360 .. 0x0361 ] | [ 0x0483 .. 0x0486 ]
extender               = 0x00B7 | 0x02D0 | 0x02D1 | 0x0387 | 0x0640 | 0x0E46 | 0x0EC6 |
                         0x3005 | [ 0x3031 .. 0x3035 ] | [ 0x309D .. 0x309E ] | [ 0x30FC ..
                         0x30FE ]
alphanum               = digit | letter
hexdigit               = [ '0' .. '9' ] | [ 'A' .. 'F' ]
not_escaped_namechar   = letter | digit | '_' | combiningchar | extender
escaped_namechar       = '.' | '-' | not_escaped_namechar
namechar               = ( escape_char escaped_namechar ) | not_escaped_namechar
reserved               = '/' | '?' | '#' | '[' | ']' | ';' | ':' | '@' | '&' | '=' | '+' | '$' | ','
mark                   = '-' | '_' | '.' | '!' | '~' | '*' | ''' | '(' | ')'
escaped                = '%' hexdigit hexdigit
unreserved             = letter | digit | mark
scheme                 = letter ( letter | digit | '+' | '-' | '.' )*
port                   = digit*
idomainlabel           = alphanum ( ( alphanum | '-' )* alphanum )?
dec_octet              = digit | ( [ 0x31 .. 0x39 ] digit ) | ( '1' digit digit ) | ( '2' [ 0x30 ..
                         0x34 ] digit ) | ( '25' [ 0x30 .. 0x35 ] )
ipv4address            = dec_octet '.' dec_octet '.' dec_octet '.' dec_octet
h4                     = hexdigit hexdigit? hexdigit? hexdigit?
ls32                   = ( h4 ':' h4 ) | ipv4address
ipv6address            = ( ( h4 ':' )* h4 )? '::' ( h4 ':' )* ls32 | ( ( h4 ':' )* h4 )? '::' h4 | (
                         ( h4 ':' )* h4 )? '::'
ipv6reference          = '[' ipv6address ']'
ucschar                = [ 0xA0 .. 0xD7FF ] | [ 0xF900 .. 0xFDCF ] | [ 0xFDF0 .. 0xFFEF ]
iunreserved            = unreserved | ucschar
ipchar                 = iunreserved | escaped | ';' | ':' | '@' | '&' | '=' | '+' | '$' | ','
isegment               = ipchar*
```

```
ipath_segments                   = isegment ( '/' isegment )*
iuserinfo                        = ( iunreserved | escaped | ';' | ':' | '&' | '=' | '+' | '$' | ',' )*
iqualified                       = ( '.' idomainlabel )* '.' ?
ihostname                        = idomainlabel iqualified
ihost                            = ( ipv6reference | ipv4address | ihostname )?
iauthority                       = ( iuserinfo '@' )? ihost ( ':' port )?
iabs_path                        = '/' ipath_segments
inet_path                        = '//' iauthority ( iabs_path )?
irel_path                        = ipath_segments
ihier_part                       = inet_path | iabs_path | irel_path
iprivate                         = [ 0xE000 .. 0xF8FF ]
iquery                           = ( ipchar | iprivate | '/' | '?' )*
ifragment                        = ( ipchar | '/' | '?' )*
iri_f                            = scheme ':' ihier_part ( '?' iquery )? ( '#' ifragment )?
absolute_iri                     = scheme ':' ihier_part ( '?' iquery )?
relative_iri                     = ihier_part ( '?' iquery )? ( '#' ifragment )?
iric                             = reserved | iunreserved | escaped
iri_reference                    = iri_f | relative_iri
tab                              = 9
cr                               = 13
lf                               = 10
eol                              = cr lf | cr | lf
squote                           = '''
dquote                           = '"'
not_cr_lf                        = [ all - [ cr+ lf ] ]
escaped_char                     = escape_char all
not_escape_char_not_dquote       = [ all - [ '"' + escape_char ] ]
string_content                   = escaped_char | not_escape_char_not_dquote
long_comment_content             = [ all - '/' ] | [ all - '*' ] '/'
long_comment                     = '/*' long_comment_content* '*/'
begin_comment                    = '//' | 'comment '
short_comment                    = begin_comment not_cr_lf* eol
comment                          = short_comment | long_comment
blank                            = ( ' ' | tab | eol )+
qmark                            = '?'
luridel                          = '_"'
ruridel                          = '"'
```

Tokens

```
t_blank              = blank
t_comment            = comment
comma                = ','
endpoint             = '.' blank
lpar                 = '('
rpar                 = ')'
lbracket             = '['
rbracket             = ']'
lbrace               = '{'
rbrace               = '}'
hash                 = '#'
t_and                = 'and'
t_or                 = 'or'
t_implies            = 'implies' | '->'
t_implied_by         = 'impliedBy' | '<-'
t_equivalent         = 'equivalent' | '<->'
t_implied_by_lp      = ':-'
t_constraint         = '!-'
t_not                = 'neg' | 'naf'
```

```
t_exists           = 'exists'
t_forall           = 'forall'
t_univfalse        = 'false'
t_univtrue         = 'true'
gt                 = '>'
lt                 = '<'
gte                = '>='
lte                = '=<'
equal              = '='
unequal            = '!='
add_op             = '+'
sub_op             = '-'
star               = '*'
div_op             = '/'
t_assumption       = 'assumption'
t_axiom            = 'axiom'
t_capability       = 'capability'
t_choreography     = 'choreography'
t_concept          = 'concept'
t_definedby        = 'definedBy'
t_effect           = 'effect'
t_endnfp           = 'endNonFunctionalProperties' | 'endnfp'
t_ggmediator       = 'ggMediator'
t_goal             = 'goal'
t_hasvalue         = 'hasValue'
t_impliestype      = 'impliesType'
t_importontology   = 'importsOntology'
t_instance         = 'instance'
t_interface        = 'interface'
t_inverseof        = 'inverseOf'
t_memberof         = 'memberOf'
t_namespace        = 'namespace'
t_nfp              = 'nonFunctionalProperties' | 'nfp'
t_oftype           = 'ofType'
t_ontology         = 'ontology'
t_oomediator       = 'ooMediator'
t_orchestration    = 'orchestration'
t_postcondition    = 'postcondition'
t_precondition     = 'precondition'
t_reflexive        = 'reflexive'
t_relation         = 'relation'
t_relation_instance = 'relationInstance'
t_sharedvariable   = 'sharedVariables'
t_source           = 'source'
t_subconcept       = 'subConceptOf'
t_subrelation      = 'subRelationOf'
t_symmetric        = 'symmetric'
t_target           = 'target'
t_transitive       = 'transitive'
t_usemediator      = 'usesMediator'
t_useservice       = 'usesService'
t_webservice       = 'webService'
t_wgmediator       = 'wgMediator'
t_wsmlvariant      = 'wsmlVariant'
t_wwmediator       = 'wwMediator'
variable           = qmark alphanum+
anonymous          = '_#'
nb_anonymous       = '_#' digit+
pos_integer        = digit+
pos_decimal        = digit+ '.' digit+
```

string	=	dquote string_content* dquote
full_iri	=	luridel iri_reference ruridel
name	=	(letter | '_') namechar*

Ignored Tokens

- t_blank
- t_comment

Productions

wsml	=	wsmlvariant? namespace? definition*
wsmlvariant	=	t_wsmlvariant full_iri
namespace	=	t_namespace prefixdefinitionlist
prefixdefinitionlist	=	{defaultns} full_iri
		| {prefixdefinitionlist} lbrace prefixdefinition moreprefixdefinitions* rbrace
prefixdefinition	=	{namespacedef} name full_iri
		| {default} full_iri
moreprefixdefinitions	=	comma prefixdefinition
definition	=	{goal} goal
		| {ontology} ontology
		| {webservice} webservice
		| {mediator} mediator
header	=	{nfp} nfp
		| {usesmediator} usesmediator
		| {importsontology} importsontology
usesmediator	=	t_usemediator idlist
importsontology	=	t_importontology idlist
nfp	=	t_nfp attributevalue* t_endnfp
mediator	=	{oomediator} oomediator
		| {ggmediator} ggmediator
		| {wgmediator} wgmediator
		| {wwmediator} wwmediator
oomediator	=	t_oomediator id? nfp? importsontology? sources? target? use_service?
ggmediator	=	t_ggmediator id? header* sources? target? use_service?
wgmediator	=	t_wgmediator id? header* source? target? use_service?
wwmediator	=	t_wwmediator id? header* source? target? use_service?
use_service	=	t_useservice id
source	=	t_source id
msources	=	t_source lbrace id moreids* rbrace
sources	=	{single} source
		| {multiple} msources
target	=	t_target id
goal	=	t_goal id? header* capability? interfaces*
webservice	=	t_webservice id? header* capability? interfaces*
capability	=	t_capability id? header* sharedvardef? pre_post_ass_or_eff*
sharedvardef	=	t_sharedvariable variablelist
pre_post_ass_or_eff	=	{precondition} t_precondition axiomdefinition
		| {postcondition} t_postcondition axiomdefinition
		| {assumption} t_assumption axiomdefinition
		| {effect} t_effect axiomdefinition
interfaces	=	{single} interface
		| {multiple} minterfaces

```
minterfaces          = t_interface lbrace id moreids* rbrace
interface            = t_interface id? header* choreography? orchestration?
choreography         = t_choreography id
orchestration        = t_orchestration id
ontology             = t_ontology id? header* ontology_element*
ontology_element     = {concept}          concept
                     | {instance}         instance
                     | {relation}         relation
                     | {relationinstance} relationinstance
                     | {axiom}            axiom
concept              = t_concept id superconcept? nfp? attribute*
superconcept         = t_subconcept idlist
att_type             = {open_world}   t_oftype
                     | {closed_world} t_impliestype
attribute            = id attributefeature* att_type cardinality? idlist nfp?
cardinality          = lpar pos_integer cardinality_number? rpar
cardinality_number   = {finite_cardinality}   pos_integer
                     | {infinite_cardinality} star
attributefeature     = {transitive} t_transitive
                     | {symmetric}  t_symmetric
                     | {inverse}    t_inverseof lpar id rpar
                     | {reflexive}  t_reflexive
instance             = t_instance id? memberof? nfp? attributevalue*
memberof             = t_memberof idlist
attributevalue       = id t_hasvalue valuelist
relation             = t_relation id arity? paramtyping? superrelation? nfp?
paramtyping          = lpar paramtype moreparamtype* rpar
paramtype            = att_type idlist
moreparamtype        = comma paramtype
superrelation        = t_subrelation idlist
arity                = div_op pos_integer
relationinstance     = t_relation_instance [name]: id? [relation]: id lpar value morevalues* rpar nfp?
axiom                = t_axiom axiomdefinition
axiomdefinition      = {use_axiom}     id
                     | {nfp_axiom}     id? nfp
                     | {defined_axiom} id? nfp? log_definition
log_definition       = t_definedby log_expr+
log_expr             = {lp_rule}          [head]: expr t_implied_by_lp [body]: expr endpoint
                     | {constraint}       t_constraint expr endpoint
                     | {other_expression} expr endpoint
expr                 = {implication} expr imply_op disjunction
                     | {disjunction} disjunction
disjunction          = {conjunction} conjunction
                     |               disjunction t_or conjunction
conjunction          = {subexpr} subexpr
                     |           conjunction t_and subexpr
subexpr              = {negated}    t_not subexpr
                     | {simple}     simple
                     | {complex}    lpar expr rpar
                     | {quantified} quantified
quantified           = quantifier_key variablelist lpar expr rpar
```

```
simple              = {molecule}     molecule
                    | {comparison}   comparison
                    | {atom}         term
molecule            = {concept_molecule_preferred}      term attr_specification? cpt_op termlist
                    | {concept_molecule_nonpreferred}   term cpt_op termlist attr_specification
                    | {attribute_molecule}              term attr_specification
attr_specification  = lbracket attr_rel_list rbracket
attr_rel_list       = {attr_relation} attr_relation
                    |                 attr_rel_list comma attr_relation
attr_relation       = {attr_def} term attr_def_op termlist
                    | {attr_val} term t_hasvalue termlist
comparison          = [left]: term comp_op [right]: term
functionsymbol      = {parametrized} id lpar terms? rpar
                    | {math}            lpar mathexpr math_op term rpar
mathexpr            = {sub} mathexpr math_op term
                    |       term
comp_op             = {gt}        gt
                    | {lt}        lt
                    | {gte}       gte
                    | {lte}       lte
                    | {equal}     equal
                    | {unequal}   unequal
cpt_op              = {memberof}      t_memberof
                    | {subconceptof}  t_subconcept
quantifier_key      = {forall}  t_forall
                    | {exists}  t_exists
attr_def_op         = {oftype}        t_oftype
                    | {impliestype}   t_impliestype
imply_op            = {implies}     t_implies
                    | {impliedby}   t_implied_by
                    | {equivalent}  t_equivalent
math_op             = {add} add_op
                    | {sub} sub_op
                    | {mul} star
                    | {div} div_op
prefix              = name hash
sqname              = {any}       prefix? name
                    | {relation}  prefix t_relation
                    | {source}    prefix t_source
iri                 = {iri}       full_iri
                    | {sqname}    sqname
id                  = {iri}                   iri
                    | {anonymous}             anonymous
                    | {universal_truth}       t_univtrue
                    | {universal_falsehood}   t_univfalse
idlist              = {id}      id
                    | {idlist}  lbrace id moreids* rbrace
moreids             = comma id
value               = {datatype}  functionsymbol
                    | {term}      id
                    | {numeric}   number
                    | {string}    string
valuelist           = {term}        value
                    | {valuelist}   lbrace value morevalues* rbrace
```

```
morevalues      = comma value
term            = {data}          value
                | {var}           variable
                | {nb_anonymous}  nb_anonymous
terms           = {term} term
                |        terms comma term
termlist        = {term} term
                |        lbrace terms rbrace
variables       = {variable} variable
                |            variables comma variable
variablelist    = {variable}      variable
                | {variable_list} lbrace variables rbrace
integer         = sub_op? pos_integer
decimal         = sub_op? pos_decimal
number          = {integer} integer
                | {decimal} decimal
```

A.2. Example of the Human-Readable Syntax

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"

namespace {_"http://www.example.org/ontologies/example#",
    dc      _"http://purl.org/dc/elements/1.1#",
    foaf _"http://xmlns.com/foaf/0.1/",
    xsd     _"http://www.w3.org/2001/XMLSchema#",
    wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
    loc     _"http://www.wsmo.org/ontologies/location#",
    oo      _"http://example.org/ooMediator#"}

/*****************************
 * ONTOLOGY
 *****************************/
ontology _"http://www.example.org/ontologies/example"
    nfp
        dc#title hasValue "WSML example ontology"
        dc#subject hasValue "family"
        dc#description hasValue "fragments of a family ontology to provide WSML examples"
        dc#contributor hasValue { _"http://homepage.uibk.ac.at/~c703240/foaf.rdf",
            _"http://homepage.uibk.ac.at/~csaa5569/",
            _"http://homepage.uibk.ac.at/~c703239/foaf.rdf",
            _"http://homepage.uibk.ac.at/homepage/~c703319/foaf.rdf" }
        dc#date hasValue _date("2004-11-22")
        dc#format hasValue "text/html"
        dc#language hasValue "en-US"
        dc#rights hasValue _"http://www.deri.org/privacy.html"
        wsml#version hasValue "$Revision: 1.4 $"
    endnfp

    usesMediator _"http://example.org/ooMediator"

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

    /*
     * This Concept illustrates the use of different styles of
     * attributes.
     */
    concept Human
        nonFunctionalProperties
            dc#description hasValue "concept of a human being"
        endNonFunctionalProperties
        hasName ofType foaf#name
        hasParent inverseOf(hasChild) impliesType Human
        hasChild impliesType Human
        hasAncestor transitive impliesType Human
        hasWeight ofType (1) _decimal
        hasWeightInKG ofType (1) _decimal
        hasBirthdate ofType (1) _date
        hasObit ofType (0 1) _date
        hasBirthplace ofType (1) loc#location
        isMarriedTo symmetric impliesType (0 1) Human
        hasCitizenship ofType oo#country
        isAlive ofType (1) _boolean
                nfp
                    dc#relation hasValue {IsAlive}
                endnfp
```

```
relation ageOfHuman (ofType Human, ofType _integer)
      nfp
           dc#relation hasValue {FunctionalDependencyAge}
      endnfp

axiom IsAlive
     definedBy
          ?x[isAlive hasValue _boolean("true")] :-
               naf ?x[hasObit hasValue ?obit] memberOf Human.
          ?x[isAlive hasValue _boolean("false")]
          impliedBy
               ?x[hasObit hasValue ?obit] memberOf Human.

 axiom FunctionalDependencyAlive
      definedBy
           !- IsAlive(?x,?y1) and
                    IsAlive(?x,?y2) and ?y1 != ?y2.

concept Man subConceptOf Human
     nfp
          dc#relation hasValue ManDisjointWoman
     endnfp

concept Woman subConceptOf Human
     nfp
          dc#relation hasValue ManDisjointWoman
     endnfp

/*
 * Illustrating general disjointness between two classes
 * via a constraint
 */
axiom ManDisjointWoman
     definedBy
          !- ?x memberOf Man and ?x memberOf Woman.

/*
 * Refining a concept and restricting an existing attribute
 */
concept Parent subConceptOf Human
     nfp
          dc#description hasValue "Human with at least one child"
     endnfp
     hasChild impliesType (1 *) Human

/*
 * Using an      axiom to define class membership and an additional
 * axiom as constraint
 */
concept Child subConceptOf Human
     nfp
          dc#relation hasValue {ChildDef, ValidChild}
     endnfp

axiom ChildDef
     nfp
          dc#description hasValue "Human being not older than 14 (the concrete
               age is an arbitrary choice and only made for illustration)"
     endnfp
     definedBy
          forall ?x (
               ?x memberOf Human and ageOfHuman(?x,?age)
               and ?age =< 14 implies ?x memberOf Child).

axiom ValidChild
     nfp
          dc#description hasValue "Note: ?x.hasAgeInYears > 14      would imply that the
               constraint is violated if the age is known to be bigger than 14;
               the chosen axiom neg ?x.hasAgeInYears =< 14 on the other hand says that
               whenever you know the age and it is less or equal 14 the constraint
               is not violated, i.e. if the age is not given the constraint is violated."
     endnfp
     definedBy
          !- ?x memberOf Child and ageOfHuman(?x,?age)
               and ?age > 14.

/*
 * Defining complete subclasses by use of axioms
 */
concept Girl subConceptOf Woman
     nfp
          dc#relation hasValue CompletenessOfChildren
     endnfp

concept Boy
     nfp
          dc#relation hasValue {ABoy,CompletenessOfChildren}
     endnfp

/*
 * This axiom implies that Boy is a Man and a Child and every Man which
 *      is also a Child is a Boy
 */
axiom ABoy
```

```
        definedBy
            forall ?x (
                ?x memberOf Boy equivalent ?x memberOf Man and ?x memberOf Child ) .

    /*
     * This axioms implies that every child has to be either a boy or a girl
     * (or both).
     * This is not the same as the axiom ManDisjointWoman, which says that
     * one cannot be man and woman at once. However, from the fact that every
     * boy is a Man and every Girl is a Woman, together with the constraint
     * ManDisjointWoman, we know that no child can be both a Girl and a Boy.
     */
    axiom CompletenessOfChildren
        definedBy
            !- ?x memberOf Child and naf (?x memberOf Girl or ?x memberOf Boy) .

    instance Mary memberOf {Parent, Woman}
        nfp
            dc#description hasValue "Mary is parent of the twins Paul and Susan"
        endnfp
        hasName hasValue "Maria Smith"
        hasBirthdate hasValue _date("1949-09-12")
        hasChild hasValue {Paul, Susan}

    instance Paul memberOf {Parent, Man}
        hasName hasValue "Paul Smith"
        hasBirthdate hasValue _date(1976,08,16)
        hasChild hasValue George
        hasCitizenship hasValue oo#de

    instance Susan memberOf Woman
        hasName hasValue "Susan Jones"
        hasBirthdate hasValue _date(1976,08,16)

     /*
        * This will be automatically an instance of Boy, since George is a
        * Man younger than 14.
        */
    instance George memberOf Man
        hasName hasValue "George Smith"
        /*hasAncestor hasValue Mary - can be inferred from the rest of this example */
        hasWeighthasWeightInKG hasValue _decimal("3.52")
        hasBirthdate hasValue _date(2004,10,21)

    relationInstance ageOfHuman(George, 1)

/****************************
 * WEBSERVICE
 ****************************/
webService _"http://example.org/Germany/BirthRegistration"
    nfp
        dc#title hasValue "Birth registration service for Germany"
        dc#type hasValue _"http://www.wsmo.org/2004/d2/#webservice"
        wsml#version hasValue "$Revision: 1.4 $"
    endnfp

    usesMediator { _"http://example.org/ooMediator" }

    importsOntology { _"http://www.example.org/ontologies/example",
        _"http://www.wsmo.org/ontologies/location" }

    capability
        sharedVariables ?child
        precondition
            nonFunctionalProperties
                dc#description hasValue "The input has to be boy or a girl
                    with birthdate in the past and be born in Germany."
            endNonFunctionalProperties
            definedBy
                ?child memberOf Child
                    and ?child[hasBirthdate hasValue ?brithdate]
                    and wsml#dateLessThan(?birthdate,wsml#currentDate())
                    and ?child[hasBirthplace hasValue ?location]
                    and ?location[locatedIn hasValue oo#de]
                    or (?child[hasParent hasValue ?parent] and
                            ?parent[hasCitizenship hasValue oo#de] ) .

        assumption
            nonFunctionalProperties
                dc#description hasValue "The child is not dead"
            endNonFunctionalProperties
            definedBy
                ?child memberOf Child
                    and naf ?child[hasObit hasValue ?x].

        effect
            nonFunctionalProperties
                dc#description hasValue "After the registration the child
                    is a German citizen"
            endNonFunctionalProperties
            definedBy
                ?child memberOf Child
                    and ?child[hasCitizenship hasValue oo#de].
```

```
    interface
        choreography _"http://example.org/exChoreograph"
        orchestration _"http://example.org/exOrchestration"

/******************************
 * GOAL
 ******************************/
goal _"http://example.org/Germany/GetCitizenShip"
    nonFunctionalProperties
        dc#title hasValue "goal of getting a citizenship within Germany"
        dc#type hasValue _"http://www.wsmo.org/2004/d2#goals"
        wsml#version hasValue "$Revision: 1.4 $"
    endNonFunctionalProperties

    usesMediator { _"http://example.org/ooMediator" }

    importsOntology { _"http://www.example.org/ontologies/example",
        _"http://www.wsmo.org/ontologies/location" }

    capability
        sharedVariables ?Human
        effect havingACitzienShip
            nonFunctionalProperties
                dc#description hasValue "This goal expresses the general
                    desire of becoming citizen of Germany."
            endNonFunctionalProperties
            definedBy
                ?Human memberOf Human[hasCitizenship hasValue oo#de] .

goal _"http://example.org/Germany/RegisterGeorge"
    nfp
        dc#title hasValue "goal of getting a Registration for Paul's son George"
        dc#type hasValue _"http://www.wsmo.org/2004/d2#goals"
        wsml#version hasValue "$Revision: 1.4 $"
    endnfp

    usesMediator { _"http://example.org/ooMediator" }

    importsOntology     { _"http://www.example.org/ontologies/example",
        _"http://www.wsmo.org/ontologies/location" }

    capability
        effect havingRegistrationForGeorge
            nfp
                dc#description hasValue "This goal expresses Paul's desire
                    for registering his son with the German birth registration board."
            endnfp
            definedBy
                George[hasCitizenship hasValue oo#de] .

//Functional description of a Web Service
webService bankTransaction
    capability
        sharedVariables {?i1,?i2}
        precondition
            definedBy
                ?i1[balance hasValue ?x] memberOf account and
                ?x >= ?i2.
        postcondition
            definedBy
                ?i1[balance hasValue ?y] and
                ?i1[balance hasValue (?y - ?i2)].

/******************************
 * MEDIATOR
 ******************************/
ooMediator _"http://example.org/ooMediator"
    nonFunctionalProperties
        dc#description hasValue "This ooMediator translates the owl
            description of the iso ontology to wsml and adds the
            necessary statements to make them memberOf loc:country
            concept of the wsmo location ontology."
        dc#type hasValue _"http://www.wsmo.org/2004/d2/#ggMediator"
        wsml#version hasValue "$Revision: 1.4 $"
    endNonFunctionalProperties
    source { _"http://www.daml.org/2001/09/countries/iso#",
        _"http://www.wsmo.org/ontologies/location"}

/*
 * This mediator is used to link the two goals. The mediator defines
 * a connection between the general goal ('GetCitizenShip') as
 * generic and reusable goal which is refined in the concrete
 * goal ('RegisterGeorge').
 */
ggMediator _"http://example.org/ggMediator"
    nonFunctionalProperties
        dc#title hasValue "GG Mediator that links the general goal of getting a citizenship
                                        with the concrete goal of registering George"
        dc#subject hasValue { "ggMediator", "Birth", "Online Birth-Registration" }
        dc#type hasValue _"http://www.wsmo.org/2004/d2/#ggMediator"
        wsml#version hasValue "$Revision: 1.4 $"
    endNonFunctionalProperties
    source _"http://example.org/GetCitizenShip"
    target _"http://example.org/RegisterGeorge"
```

```
/*
 * In the general case the generic goal and the WS are known before a concrete
 * request is made and can be statically linked, to avoid reasoning during
 * the runtime of a particular request. The fact that the WS fullfills at
 * least partially the goal it is explicitly stated in the wgMediator.
 */
wgMediator _"http://example.org/wgMediator"
    nonFunctionalProperties
        dc#type hasValue _"http://www.wsmo.org/2004/d2/#wgMediator"
    endNonFunctionalProperties
    source _"http://example.org/BirthRegistration"
    target _"http://example.org/GetCitizenShip"
```

Appendix B. Schemas for the XML Exchange Syntax

In the following sections we present the XML Schemas for the XML syntax of WSML, which was introduced in Chapter 9.
The schemas are available online at http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/wsml-xml-syntax.xsd.

This schema includes two module schemas:

- WSML identifiers (http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/wsml-identifiers.xsd)
- Logical expressions of WSML (http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/wsml-expr.xsd).

Furthermore, the schema imports an additional schema for the basic Dublin Core elements (http://dublincore.org/schemas/xmls/qdc/2003/04/02/dc.xsd).

Userfriendly documentation for the schemas is available from the following locations:

- *XML syntax for WSML:*
  http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/ documentation/wsml-xml-syntax.xsd.html
- *XML syntax for WSML identifiers:*
  http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/ documentation/wsml-identifiers.xsd.html
- *XML syntax for WSML logical expressions:*
  http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/ documentation/wsml-expr.xsd.html

Appendix C. Datatypes and Built-ins in WSML

The appendix contains a preliminary list of built-in functions and relations for datatypes in WSML. Furthermore, it will contain a translation of syntactic shortcuts to datatype predicates.

Appendix C.1. WSML Datatypes

WSML recommends the use of XML Schema datatypes as defined in [Biron & Malhorta, 2004] for the representation of concrete values, such as strings and integers. WSML defines a number of built-in functions for the use of XML Schema datatypes.

WSML allows direct usage of the string, integer and decimal data values in the language. These values have a direct correspondence with values of the XML Schema datatypes *string*, *integer*, and *decimal*, respectively. Values of these most primitive datatypes can be used to construct values of more complex datatypes. Table C.1 lists datatypes allowed in WSML with the name of the datatype constructor, the name of the corresponding XML Schema datatype, a short description of the datatype (corresponding with the value space as defined in [Biron & Malhorta, 2004]) and an example of the use of the datatype.

WSML datatype constructor	XML Schema datatype	Description of the Datatype	Syntax
_string	string	A finite-length sequence of Unicode characters, where each occurrence of the double quote '"' is escaped using the backslash symbol: '\"' and the backslash is escaped using the backslash: '\\'.	_string("*any-character**")
_decimal	decimal	That subset of natural numbers which can be represented with a finite sequence of decimal numerals.	_decimal("'-'?*numeric*+.*numeric*+")
_integer	integer	That subset of decimals which corresponds with natural numbers.	_integer("'-'?*numeric*+")
_float	float		_float("*see XML Schema document*")
_double	double		_double("*see XML Schema document*")
_iri	similar to anyURI	An IRI conforms to [Duerst & Suignard, 2005]. Every URI is an IRI.	_iri("*iri-according-to-rfc3987*")
_sqname	serialized QName	An sQName is a pair {namespace, localname}, where the localname us concatenated to the namespace to form an IRI. An sQName is actually equivalent to the IRI which results from concatenating the namespace and the localname.	_sqname("*iri-according-to-rfc3987*", "*localname*")
_boolean	boolean		_boolean("*true-or-false*")
_duration	duration		_duration(*year*, *month*, *day*, *hour*, *minute*, *second*)
_dateTime	dateTime		_dateTime(*year*, *month*, *day*, *hour*, *minute*, *second*, *timezone-hour*, *timezone-minute*) _dateTime(*year*, *month*, *day*, *hour*, *minute*, *second*)
_time	time		_time(*hour*, *minute*, *second*, *timezone-hour*, *timezone-minute*) _time(*hour*, *minute*, *second*)
_date	date		_date(*year*, *month*, *day*, *timezone-hour*, *timezone-minute*) _date(*year*, *month*, *day*)
_gyearmonth	gYearMonth		_gyearmonth(*year*, *month*)
_gyear	gYear		_gyear(*year*)
_gmonthday	gMonthDay		_gmonthday(*month*, *day*)
_gday	gDay		_gday(*day*)
_gmonth	gMonth		_gmonth(*month*)
_hexbinary	hexBinary		_hexbinary(*hexadecimal-encoding*)
_base64binary	base64Binary		_base64binary(*hexadecimal-encoding*)

Table C.1: WSML Datatypes

Table C.2 contains the shortcut syntax for the string, integer, decimal, IRI and sQName datatypes.

WSML datatype constructor	Shortcut syntax	Example
_string	"*any-character**"	"John Smith"
_decimal	'-'?*numeric*+.*numeric*+	4.2, 42.0
_integer	'-'?*numeric*+	42, -4
_iri	_"*iri-according-to-rfc3987*"	_"http://www.wsmo.org/wsml/wsml-syntax#"

Table C.2: WSML Datatype shortcut syntax

WSML datatype constructor	Shortcut syntax	Example
_sqname	alphanumeric+'#'alphanumeric+	wsml#concept, xsd#integer

Appendix C.2. WSML Datatype Predicates

This section contains a list of datatype predicates suggested for use in WSML. 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.

The current list is only based on the built-in support in the WSML language through the use of special symbols. Find a translation of the built-in symbols to datatype predicates in the next section. The symbol 'range' signifies the range of the function. Functions in XQuery have a defined range, whereas predicates only have a domain. Therefore, the first argument of a WSML datatype predicate which represents a function represents the range of the function. Comparators in XQuery are functions, which return a boolean value. These comparators are directly translated to predicates. If the XQuery function returns 'true', the arguments of the predicate are in the extension of the predicate. See Table C.3 for the complete list. In the evaluation of the predicates, the parameters 'A' and 'B' must be bound; the parameter 'range' does not need to be bound. A parameter is bound if it is substituted with a value. It is not bound if it is substituted with a variable. The variable is then used to convey some outcome of the function.

WSML datatype predicate	XQuery function	Datatype (A)	Datatype (B)	Return datatype
wsml#numericEqual(A,B)	op#numeric-equal(A,B)	numeric	numeric	
wsml#numericGreaterThan(A,B)	op#numeric-greater-than(A,B)	numeric	numeric	
wsml#numericLessThan(A,B)	op#numeric-less-than(A,B)	numeric	numeric	
wsml#stringEqual(A,B)	op#numeric-equal(fn:compare(A, B), 1)	xsd#string	xsd#string	
wsml#numericAdd(range,A,B)	op#numeric-add(A,B)	numeric	numeric	numeric
wsml#numericSubtract(range,A,B)	op#numeric-subtract(A,B)	numeric	numeric	numeric
wsml#numericMultiply(range,A,B)	op#numeric-multiply(A,B)	numeric	numeric	numeric
wsml#numericDivide(range,A,B)	op#numeric-divide(A,B)	numeric	numeric	numeric

Table C.3: WSML Datatype Predicates

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.

Appendix C.3. Translating Built-in Symbols to Datatype Predicates

In this section, we provide the translation of the built-in (function and predicate) symbols for datatype predicates to these datatype predicates.

We distinguish between built-in functions and built-in relations. Functions have a defined domain and range. Relations only have a domain and can in fact be seen as functions, which return a boolean, as in XPath/XQuery [Malhotra et al., 2004]. We first provide the translation of the built-in relations and then present the rewriting rules for the built-in functions.

The following table provides the translation of the built-in relations:

Operator	Datatype (A)	Datatype (B)	Predicate
A = B	string	string	wsml#stringEqual(A,B)
A != B	xsd#string	xsd#string	wsml#stringInequal(A,B)
A = B	numeric	numeric	wsml#numericEqual(A,B)
A != B	numeric	numeric	wsml#numericInequal(A,B)
A < B	numeric	numeric	wsml#lessThan(A,B)
A =< B	numeric	numeric	wsml#lessEqual(A,B)
A > B	numeric	numeric	wsml#greaterThan(A,B)
A >= B	numeric	numeric	wsml#greaterEqual(A,B))

Table C.4: WSML infix operators and corresponding datatype predicates

We list the built-in functions and their translation to datatype predicates in Table C.5. In the table, ?x1 represents a unique newly introduced variable, which stands for the range of the function.

Operator	Datatype (A)	Datatype (B)	Predicate
A + B	numeric	numeric	wsml#numericAdd(?x1,A,B)
A - B	numeric	numeric	wsml#numericSubtract(?x1,A,B)
A * B	numeric	numeric	wsml#numericMultiply(?x1,A,B)
A / B	numeric	numeric	wsml#numericDivide(?x1,A,B)

Table C.5: Translation of WSML infix operators to datatype predicates

Function symbols in WSML are not as straightforward to translate to datatype predicates as are relations. However, if we see the predicate as a function, which has the range as its first argument, we can introduce a new variable for the return value of the function and replace an occurrence of the function symbol with the newly introduced variable and append the newly introduced predicate to the conjunction of which the top-level predicate is part.

Formulas containing nested built-in function symbols can be rewritten to datatype predicate conjunctions according to the following algorithm:

1. Select an atomic occurrence of a datatype function symbol. An atomic occurrence is an occurrence of the function symbol with only identifiers (which can be variables) as arguments.
2. Replace this occurrence with a newly introduced variable and append to the conjunction of which the function symbols is part the datatype predicate, which corresponds with the function symbol where the first argument (which represents the range) is the newly introduced variable.
3. If there are still occurrences of function symbols in the formula, go back to step (1), otherwise, return the formula.

We present an example of the application of the algorithm to the following expression:

```
?w = ?x + ?y + ?z
```

We first substitute the first occurrence of the function symbol '+' with a newly introduced variable ?x1 and append the predicate wsml#numericAdd(?x1, ?x, ?y) to the conjunction:

```
?w = ?x1 + ?z and wsml#numericAdd(?x1, ?x, ?y)
```

Then, we substitute the remaining occurrence of '+' accordingly:

```
?w = ?x2 and wsml#numericAdd(?x1, ?x, ?y) and wsml#numericAdd(?x2, ?x1, ?z)
```

Now, we don't have any more built-in function symbols to substitute and we merely substitute the built-in relation '=' to obtain the final conjunction of datatype predicates:

```
wsml#numericEqual(?w, ?x2) and wsml#numericAdd(?x1, ?x, ?y) and wsml#numericAdd(?x2, ?x1, ?z)
```

Appendix D. WSML Keywords

This appendix lists all WSML keywords, long with the section of the deliverable where they have been described. Keywords are differentiated per WSML variant. Keywords common to all WSML variants are referred to under the column "Common element". The elements specific to a certain variant are listed under the specific variant. A '+' in the table indicates that the keyword is included is inherited from the lower variant. Finally, a reference to a specific section indicates that the definition of the keyword can be found in this section.

Note that there are two layerings in WSML. Both layerings are complete syntactical and semantic (wrt. entailment of ground facts) layerings. The first layering is WSML-Core > WSML-Flight > WSML-Rule > WSML-Full, with WSML-Core being the lowest language in the layering and WSML-Full being the highest. This means, among other things, that every keyword of WSML-Core is a keyword of WSML-Flight, every keyword of WSML-Flight is a keyword of WSML-Rule, etc. The second layering is WSML-Core > WSML-DL > WSML-Full, which means that every WSML-Core keyword is a WSML-DL keyword, etc. Note that the second layering is a complete semantic layering, also with respect to entailment of non-ground formulae. For now we do not take WSML-DL into account in the table.

It can happen that the definition of a specific WSML element of a lower variant is expanded in a higher variant. For example, concept definitions in WSML-Flight are an extended version of concept definitions in WSML-Core.

We list all keywords of the WSML conceptual syntax in Table D.1.

Keyword	Section	Core	Flight	Rule	Full
wsmlVariant	2.2.1	+	+	+	+
namespace	2.2.2	+	+	+	+
nonFunctionalProperties	2.2.3	+	+	+	+
endNonFunctionalProperties	2.2.3	+	+	+	+
nfp	2.2.3	+	+	+	+
endnfp	2.2.3	+	+	+	+
importsOntology	2.2.3	+	+	+	+
usesMediator	2.2.3	+	+	+	+
ontology	2.3	+	+	+	+
concept	2.3.1	+	+	+	+
subConceptOf	2.3.1	+	+	+	+
ofType	2.3.1.1	+	+	+	+
impliesType	2.3.1.1	+	+	+	+
transitive	2.3.1.1	+	+	+	+
symmetric	2.3.1.1	+	+	+	+
inverseOf	2.3.1.1	+	+	+	+
reflexive	2.3.1.1		+	+	+
relation	2.3.2	+	+	+	+
subRelationOf	2.3.2	+	+	+	+
instance	2.3.3	+	+	+	+
memberOf	2.3.3	+	+	+	+
hasValue	2.3.3	+	+	+	+
relationInstance	2.3.3	+	+	+	+
axiom	2.3.4	+	+	+	+

Table D.1: WSML keywords

Keyword	Section	Core	Flight	Rule	Full
definedBy	2.3.4	+	+	+	+
capability	2.4.1	+	+	+	+
sharedVariables	2.4.1	+	+	+	+
precondition	2.4.1	+	+	+	+
assumption	2.4.1	+	+	+	+
postcondition	2.4.1	+	+	+	+
effect	2.4.1	+	+	+	+
interface	2.4.2	+	+	+	+
choreography	2.4.2	+	+	+	+
orchestration	2.4.2	+	+	+	+
goal	2.5	+	+	+	+
ooMediator	2.6	+	+	+	+
ggMediator	2.6	+	+	+	+
wgMediator	2.6	+	+	+	+
wwMediator	2.6	+	+	+	+
source	2.6	+	+	+	+
target	2.6	+	+	+	+
usesService	2.6	+	+	+	+
webService	2.7	+	+	+	+

Table D.2 lists the keywords allowed in logical expressions. The complete logical expression syntax is defined in Section 2.8. Besides the keywords in this list, WSML also allows for the use of a number of infix operators for built-in predicates, see also Appendix C

Keyword	WSML-Core	WSML-Flight	WSML-Rule	WSML-Full
true	+	+	+	+
false	+	+	+	+
memberOf	+	+	+	+
hasValue	+	+	+	+
subConceptOf	+	+	+	+
ofType	+	+	+	+
impliesType	+	+	+	+
and	+	+	+	+
or	+	+	+	+
implies	+	+	+	+
impliedBy	+	+	+	+
equivalent	+	+	+	+
neg	-	-	-	+
not	-	+	+	+
forall	+	+	+	+
exists	-	-	-	+

Table D.2: WSML logical expression keywords

Appendix E. Relation to WSMO Conceptual Model

WSML aims at providing a complete language to describe all elements in the WSMO conceptual model as defined in [Roman *et al.* 2004]. However, although most elements in WSMO have direct correspondences in WSML language constructs there are slight differences due to the fact that WSML is in principle a logical description language rather than a conceptual model.

Relations. The WSMO conceptual model defines relations as parts of ontologies. This concept also exists in WSML but there are slight differences. First, parameters in relations are not named in WSML. n-ary relations in a mathematical or logical sense are usually presented as a set of n-tuples which corresponds to relations in WSML. Relations with named attributes in the sense of relational databases have a strong correspondence to this view of relations being "flat" concepts which can also be represented as sets of tuples. In WSML however, one would rather model relations with named parameters as concepts with corresponding attributes. This reflects the common correspondence between conceptual modeling and the relational data model. On the contrary, n-ary relations in WSML rather correspond to n-ary predicate symbols in a logical sense.

Functions. The WSMO conceptual model defines Functions as parts of ontologies. These are defined as a special type of relations with a distinct range parameter representing the function value. We can define such functional dependencies via axioms in WSML. For instance, the age of a person id uniquely determined by the birth date of a person. We can define the computation of this function and the respective functional dependency by two axioms as follows:

```
relation ageOfHuman/2 (ofType Human, ofType xsd#integer)
    nfp

        dc#relation hasValue {AgeOfHuman, FunctionalDependencyAge}
    endnfp

axiom AgeOfHuman
    definedBy
        forall ?x,?y,?z (
            AgeOfHuman(?x,?y) equivalent

                ?x memberOf Human and
                wsml#years\-from\-duration(?y,?z) and
                ?x[hasBirthdate hasValue ?birth] and
                wsml#subtract\-dateTimes\-yielding\-dayTimeDuration(
                        ?z,
                        ?birth,
                        wsml#current\-dateTime())
        ) .

 axiom FunctionalDependencyAge
    definedBy

        forall ?x,?y1,?y2 (
                false impliedBy AgeOfHuman(?x,?y1) and

                    AgeOfHuman(?x,?y2) and ?y1 != ?y2)
```

Furthermore, any attribute with a maximal cardinality of 1 is a function.

Value Ranges of Attributes. In the WSMO conceptual model, each attribute can have assigned a single Range. Similarly relation parameters have a single Range. WSML makes a more fine grained disctinction here reflecting the different constraining and defining views of specifying the range of an attribute/parameter by the keywords *a1* **ofType** *range* and *a2* **impliesType** *range*. The former states that whenever the value of an attribute *a1* is specified in an instance, it is checked whether this attribute is a member of the range class, corresponding to an integrity constraint, whereas the latter correxsponds to an axiom inferring membership of *range* for any value of the attribute *a2*. Furthermore, WSMO allows you to specify a list of ranges on the right-hand-side of **ofType** (and **impliesType**, resp.) which simply corresponds to specifying the range specification to the intersection of the given ranges.

Defined concepts and relations. In the WSMO conceptual model, concepts and relations can have a definition assigned by the hasDefinition attribute. This attribute than contains a logical expression defining the respective concept or relation. Since a concept or relation description in WSML boils down to a set of axioms itself semantically (as shown in Section 8), definitional axioms are not directly reflected in the WSML syntax. Rather, the user is expected to specify these definitions in separate **axiom** descriptions. Basically, the possibility to have a definition via axioms directly associated with the description of the respective concept or relation, can in WSML be expressed by using the dc:relation element of the non-functional properties of a concept or relation. Here you can explicitly refer to related axioms by their identifiers.

Choreography and orchestration Interfaces. The language for defining complex interfaces of Web services is not yet defined within WSML. however, we expect it to be based on the initial drafts in [Roman & Scicluna, 2005a],[Roman & Scicluna, 2005b] where we aim at refining the language and semantics used there to a more

user-friendly level.

Appendix F. Changelog

Compared with the previous version of this document (2005-03-05), the following changes have been made:

The major changes in the content are:

- Modules have been removed
- The WSML-Core logical expression definition has been improved to make more legible
- The mapping to OWL has been improved to create a formal mapping function
- All examples have been updated to reflect the new syntax
- idlists are allowed in memberOf, subConceptOf, ofType and impliesType molecules in the logical expression syntax
- attribute features transitive, symmetric, inverseOf, have been removed from WSML-Core, because they do not fall in the Description Logic fragment.

Major future work (for v0.3):

- Specification of WSML-DL, which will syntactically layer on top of WSML-Core, but will be based semantically on OWL DL.
- Specification of WSML-Full once the approach for defining the semantics is clear
- See if the WSML semantics (Chapter 8) can be written down in a more concise way

References

[Baader et al., 2003] F. Baader, D. Calvanese, and D. McGuinness: *The Description Logic Handbook*, Cambridge University Press, 2003.

[Bray et al., 1999] T. Bray, D. Hollander, A. Layman (eds.): *Namespaces in XML*, W3C Recommentation, available from http://www.w3.org/TR/REC-xml-names/.

[Berglund et al., 2004] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J Siméon (eds.): *XML Path Language (XPath) 2.0*, W3C Working Draft, available from http://www.w3.org/TR/xpath20/.

[Berners-Lee at el., 1998] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. *Uniform resource identifiers (URI): Generic syntax*. RFC 2396, Internet Engineering Task Force, 1998.

[Biron & Malhorta, 2004] P.V. Biron and A. Malhorta. *XML Schema Part 2: Datatypes Second Edition*. W3C Recommendation 28 October 2004.

[Bonner & Kifer, 1998] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117-166. Kluwer Academic Publishers, March 1998.

[Brickley & Guha, 2004] D. Brickley and R. V. Guha. *RDF vocabulary description language 1.0: RDF schema*. W3C Recommendation 10 February 2004. Available from http://www.w3.org/TR/rdf-schema/.

[de Bruijn et al., 2004] J. de Bruijn, A. Polleres, R. Lara and D. Fensel. *OWL*$^{-}$. Deliverable d20.1v0.2, WSML, 2004. http://www.wsmo.org/TR/d20/d20.1/v0.2/

[de Bruijn et al., 2004a] J. de Bruijn, A. Polleres, R. Lara and D. Fensel. *OWL Flight*. Deliverable d20.3v0.1, WSML, 2004. http://www.wsmo.org/TR/d20/d20.3/v0.1/

[de Bruijn, 2004] J. de Bruijn (Ed). *WSML Reasoner Implementation*. WSML Working Draft D16.2v0.1, 2004. Available from http://www.wsmo.org/TR/d16/d16.2/v0.2/.

[de Bruijn & Polleres, 2004] J. de Bruijn and A. Polleres. *Towards and ontology mapping language for the semantic web*. Technical Report DERI-2004-06-30, DERI, 2004. Available from http://www.deri.org/publications/techpapers/documents/DERI-TR-2004-06-30.pdf.

[de Bruijn et al., 2005] J. de Bruijn, A. Polleres, R. Lara and D. Fensel. *OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning for the Semantic Web*. Fourteenth International World Wide Web Conference (WWW2005), 2005. To appear.

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

[Chen et al., 1993] W. Chen, M. Kifer, and D. S. Warren: HILOG: A foundation for higher-order logic programming. Journal of Logic Programming, 15(3):187-230, 1993.

[Dean & Schreiber, 2004] M. Dean, G. Schreiber, (Eds.). *OWL Web Ontology Language Reference*, W3C Recommendation, 10 February 2004. Available from http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[Eiter et al., 1997] T. Eiter, G. Gottlob, and H. Mannila: Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364-418, 1997.

[Eiter et al., 2004] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proc. of the International Conference of Knowledge Representation and Reasoning (KR04)*, 2004.

[Enderton, 2002] H. B. Enderton. *A Mathematical Introduction to Logic (2nd edition)*. Academic Press, 2002

[Fensel et al., 2001] D. Fensel, F. van Harmelen, I. Horrocks, D.L. McGuinness, and P.F. Patel- Schneider: OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16:2, May 2001.

[van Gelder et al., 1991] A. van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620-650, 1991.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070-1080, Cambridge, Massachusetts, 1988. The MIT Press.

[Ghidini and Serafini, 2000]C. Ghidini and L. Serafini: Distributed First order logics. In D.M. Gabbay and M. De Rijke, editors, *Frontiers of Combining Systems 2*, Studies in Logic and Computation, No. 7, Research Studies Press Ltd. Baldock, Hertfordshire, England, UK, 2000, pp. 121-139.

[Grosof et al., 2003] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 48-57. ACM, 2003.

[Horrocks et al., 2004] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. W3C Member Submission 21 May 2004. Available from http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

[Keller et al., 2004] U. Keller, R. Lara, and A. Polleres, *eds. WSMO Discovery*. WSMO Working Draft D5.1v0.1, 2004. Available from http://www.wsmo.org/TR/d5/d5.1/v0.1/.

[Kifer et al., 1995] M. Kifer, G. Lausen, and J. Wu: Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741-843, July 1995.

[Lara et al., 2005] R. Lara, A. Polleres, H. Lausen, D. Roman, J. de Bruijn, and D. Fensel. *A Conceptual Comparison between WSMO and OWL-S*. WSMO Deliverable D4.1v0.1, 2005. http://www.wsmo.org/2004/d4/d4.1/v0.1/

[Levy &Rousset, 1998] A.Y. Levy and M.-C. Rousset. Combining horn rules and description logics in CARIN. *Artificial Intelligence*, 104:165-209, 1995.

[Lloyd, 1987] J. W. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer-Verlag, 1987.

[Lloyd and Topor, 1984] John W. Lloyd and Rodney W. Topor. Making prolog more expressive. *Journal of Logic Programming*, 1(3):225{240, 1984.

[Malhotra et al., 2004] A. Malhotra, J. Melton, N. Walsh: *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Working Draft, available at http://www.w3.org/TR/xpath-functions/.

[Marek and Truszczynski, 1993] V. W. Marek and M. Truszczynski. *Nonmonotonic Logic: Context-dependent Reasoning*. Artificial Intelligence. Springer-Verlag, Berlin Heidelberg, 1993.

[Mitra et al., 2000] P. Mitra, G. Wiederhold, and M.L. Kersten. A graph-oriented model for articulation of ontology interdependencies. In *In Proceedings of Conference on Extending Database Technology (EDBT 2000)*, Konstanz, Germany, March 2000.

[OWL-S, 2004] The OWL Services Coalition. *OWL-S 1.1*, 2004. Available from http://www.daml.org/services/owl-s/1.1B/.

[Pan and Horrocks, 2004] J. Z. Pan and I. Horrocks, I.: *OWL-E: Extending OWL with expressive datatype expressions*. IMG Technical Report IMG/2004/KR-SW-01/v1.0, Victoria University of Manchester, 2004. Available from http://dl-web.man.ac.uk/Doc/IMGTR-OWL-E.pdf.

[Patel-Schneider et al., 2004] P. F. Patel-Schneider, P. Hayes, and I. Horrocks: *OWL web ontology language semantics and abstract syntax*. Recommendation 10 February 2004, W3C, 2004. Available from http://www.w3.org/TR/owl-semantics/.

[Przymusinski, 1989] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167-205, 1989.

[RDF] Resource Description Framework (RDF) http://www.w3.org/RDF/.

[Reiter, 1987] Raymond Reiter. A logic for default reasoning. In *Readings in Nonmonotonic Reasoning*, pages 68-93. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[Roman et al., 2004] D. Roman, H. Lausen, and U. Keller (eds.): *Web Service Modeling Ontology - Standard (WSMO - Standard)*, WSMO deliverable D2 version 1.1. available from http://www.wsmo.org/TR/d2/v1.1/.

[Roman & Scicluna, 2005a] D. Roman, J. Scicluna (eds.): *Choreography in WSMO*, WSMO deliverable D14 version 0.1. available from http://www.wsmo.org/2005/d14/v0.1/.

[Roman & Scicluna, 2005b] D. Roman, J. Scicluna (eds.): *Orchestration in WSMO*, WSMO deliverable D15 version 0.1. available from http://www.wsmo.org/2005/d15/v0.1/.

[SableCC] The SableCC Compiler Compiler. http://www.sablecc.org/

[Stollberg et al., 2004] M. Stollberg, H. Lausen, A. Polleres, and R. Lara (eds.): *WSMO Use Case Modeling and*

Testing, WSMO d3.2v0.1. available from http://www.wsmo.org/2004/d3/d3.2/.

[Weibel et al. 1998] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf: *RFC 2413 - Dublin Core Metadata for Resource Discover*y, September 1998.

[XML-NAMESPACES-1.1] T. Bray, D. Hollander, A. Layman, R. Tobin, Editors. Namespaces in XML 1.1. W3C Recommendation 04 February 2004. http://www.w3.org/TR/xml-names11/.

[Yang & Kifer, 2003] G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the semantic web. *Journal on Data Semantics*, 1:69-97, 2003.

[Yang et al., 2003] G. Yang, M. Kifer, and C. Zhao. FLORA-2: A rule- based knowledge representation and inference infrastructure for the semantic web. In *Proceedings of the Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE)*, Catania, Sicily, Italy, 2003.

Acknowledgement

The work is funded by the European Commission under the projects DIP, Knowledge Web, InfraWebs, SEKT, SWWS, ASG and Esperonto; by Science Foundation Ireland under the DERI-Lion project; by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects RW^2 and TSC.

The editors would like to thank to all the members of the WSML working group for their advice and input into this document. We would especially like to thank Douglas Foxvog and Eyal Oren for their work on deliverables superseded by this deliverable.

Footnotes

[1]The work presented in [Levy &Rousset, 1998] might serve as a starting point to define a subset of WSML-Full which could be used to enable a higher degree of interoperation between Description Logics and Logic Programming (while retaining decidability, but possibly losing tractability) than through their common core described in WSML-Core. If we would choose to minimize the interface between both paradigms, as described in [Eiter et al., 2004], it would be sufficient to add a simple syntactical construct to the Logic Programming language. This construct would stand for a query to the Description Logic knowledge base. Thus, the logic programming engine should have an interface to the Description Logic reasoner to issue queries and retrieve results.

[2]The complexity of query answering for a language with datatype predicates depends on the time required to evaluate these predicates. Therefore, when using datatypes, the complexity of query answering may grow beyond polynomial time in case evaluation of datatype predicates is beyond polynomial time.

[3]The only expressivity added by the logical expressions over the conceptual syntax is the complete class definition, and the use of individual values.