



D16.1v1.0 WSML Language Reference

WSML Final Draft 2008-08-08

This version

<http://www.wsmo.org/TR/d16/d16.1/v1.0/20080808/>

Latest version

<http://www.wsmo.org/TR/d16/d16.1/v1.0/>

Previous version

<http://www.wsmo.org/TR/d16/d16.1/v1.0/20080728/>

Editors:

Nathalie Steinmetz
Ioan Toma

Authors:

[The WSML working group members](#)

Copyright © 2008 by the authors.

This document is part of the specification of the Web Service Modeling Language (WSML), version 1.0.

The specification of WSML 1.0 consists of the following four documents.

- [WSML Language Reference](#)
- [WSML Abstract Syntax and Semantics](#)
- [WSML/XML](#)
- [WSML/RDF](#)

Abstract

WSML is a language for modeling Web services, ontologies, and related aspects, and is based on the Web Service Modeling Ontology WSMO. The formal grounding of the language is based on a number of logical formalisms, namely, Description Logics, First-Order Logic and Logic Programming. Besides providing its own language for modeling ontologies, it allows the import and use of RDF Schema and OWL ontologies for Web service description.

This document provides a reference of the features of the WSML language. It is intended for users who want to model web services and ontologies using WSML, and implementers who want to build tools based on the WSML language.

Please note that the current version of the WSML specification (that is WSML v1.0) is not backward compatible to the previous version of WSML (that is WSML v0.21). For details please refer to the [changelog](#).

Table of Contents

1. Introduction

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 Prologue
 - 2.2.1 WSML Variant Declaration
 - 2.2.2 Namespace References
 - 2.2.3 Header
- 2.3 Ontologies in WSML
 - 2.3.1 Concepts
 - 2.3.2 Instances
 - 2.3.3 Relations
 - 2.3.4 RelationInstances
 - 2.3.5 Axioms
- 2.4 Capabilities, Interfaces and Non-functional Properties in WSML
 - 2.4.1 Capabilities
 - 2.4.2 Interfaces
 - 2.4.3 Non-functional properties
- 2.5 Goals in WSML
- 2.6 Web Services in WSML
- 2.7 Mediators in WSML
 - 2.7.1 ooMediators
 - 2.7.2 wwMediators
 - 2.7.3 ggMediators
 - 2.7.4 wgMediators
- 2.8 General Logical Expressions in WSML

3 WSML-Core

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

4 WSML-DL

- [4.1. WSML-DL Logical Expression Syntax](#)

[5 WSML-Flight](#)

- [5.1. WSML-Flight Logical Expression Syntax](#)
- [5.2. 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](#)

[Appendix A. Human-Readable Syntax](#)

- [A.1. BNF-Style Grammar](#)
- [A.2. Example of Human-Readable Syntax](#)

[Appendix B. Built-ins in WSML](#)

- [B.1. WSML Datatypes](#)
- [B.2. WSML Built-in Predicates](#)
- [B.3. Translating Built-in Symbols to Predicates](#)

[Appendix C. WSML Keywords](#)

[Appendix D. Changelog](#)

[References](#)

[Acknowledgements](#)

1. Introduction

The Web Service Modeling Ontology WSMO [[Roman *et al.* 2006](#)] 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. The WSML Abstract Syntax and Semantics [[de Bruijn, 2008c](#)] closely follows the conceptual model of WSMO version 1.3 in presenting the abstract syntax as well as the semantics of WSML, and only diverges where necessary. That document also features a mapping from the abstract to the surface syntax of WSML. The latter is defined in the WSML Language Reference, i.e. in this document. It provides a specification of all WSML variants in terms of a human-readable syntax with keywords similar to the elements of the WSMO conceptual model. The WSML surface syntax might not be suitable for exchange between automated agents. Therefore, to increase the interoperability of WSML, XML and RDF-based exchange syntaxes have been developed in the following

documents:

WSML/XML:

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. A complete XML syntax for WSML is defined in [\[Toma, 2008\]](#).

WSML/RDF:

An alternative exchange syntax for WSML is WSML/RDF: it can be used to leverage the currently existing RDF tools, such as triple stores, and to syntactically combine WSML/RDF descriptions with other RDF descriptions. WSML/RDF is fully defined in [\[de Bruijn, 2008b\]](#).

Furthermore a mapping from WSML-DL to OWL DL and vice versa has been developed in the following document: [\[Steinmetz, 2008\]](#). This mapping can be seen as guide for people that are interested in conversions between WSML and OWL.

Semantic Web service descriptions refer to ontologies for the definition of the terminology used in descriptions and the specification of background knowledge. WSML includes a language for the specification of such ontologies. On the Semantic Web, the most prominent ontology languages are RDFS [\[Brickley & Guha, 2004\]](#) and OWL DL [\[Dean & Schreiber, 2004\]](#). WSML allows the use of any of these three ontology languages for the definition of terminology and background knowledge used in Semantic Web service descriptions. In addition, it is possible to import RDFS and OWL DL ontologies into WSML ontologies. See [Section 2.2.3](#) for more details.

The Rule Interchange Format Basic Logic Dialect (RIF-BLD) [\[Boley & Kifer, 2008\]](#) is a logic-based rule language for the Web, close in spirit to the Rule variant of WSML, bearing in mind that it does not include negation. RIF-BLD descriptions can be viewed as background knowledge that may be exploited in Semantic Web service descriptions. In addition, RIF-BLD rules may reuse or extend terminologies defined by RDFS or OWL ontologies [\[de Bruijn, 2008a\]](#). It is expected that future versions of WSML will allow importing RIF-BLD documents, as well as extensions of RIF-BLD, similar to the import of RDFS and OWL DL ontologies.

Please note that the current version of the WSML specification (that is WSML v0.3) is not backward compatible to the previous version of WSML (that is WSML v0.21) with respect to the following items:

- the syntax used to specify annotations and non functional properties.
- the (now forbidden) usage of relations in WSML-Core and WSML-DL.
- the removal of strong equality (`:=:`).
- the removal of the datatypes `_iri` and `_sqname`.

For more details on all changes from WSML v0.21 to WSML v1.0 please see Appendix D, the [Changelog](#).

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.

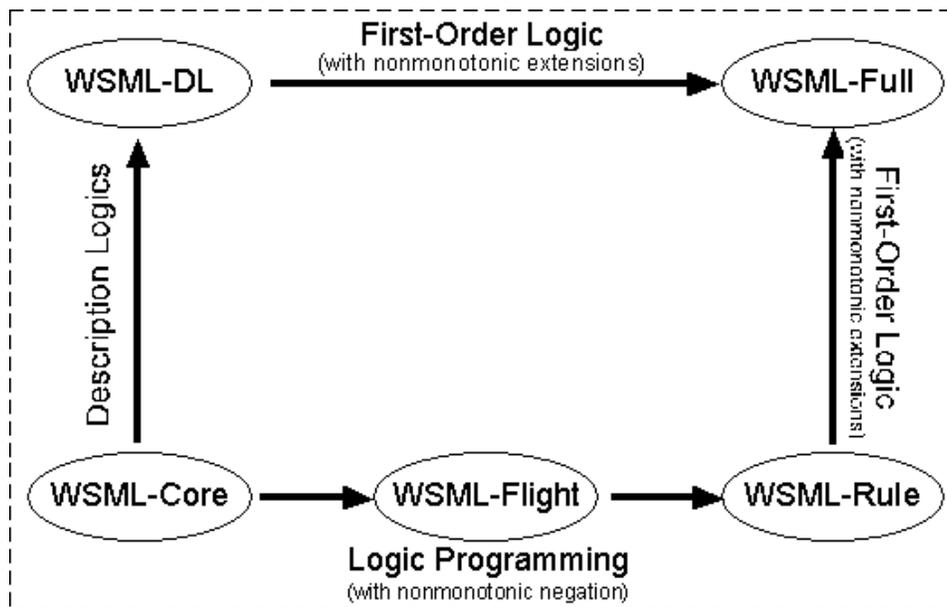


Figure 1. WSML Space

WSML-Core

This language is defined by the intersection of Description Logic and Horn Logic, based on Description Logic Programs [Grosf 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 [Grosf 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]

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, and does not require stratification of negation. The semantics for negation is based on the Stable Model Semantics [Gelfond & Lifschitz, 1988].

WSML-Full

WSML-Full unifies WSML-DL and WSML-Rule under a First-Order syntactic umbrella with extensions to support the nonmonotonic negation of WSML-Rule. There are various candidates for the specification of the semantics of WSML Full, both as there is no widespread consensus about what is the right formalism for combining Description Logic ontologies with nonmonotonic rules, WSML does not specify a semantics for its Full variant.

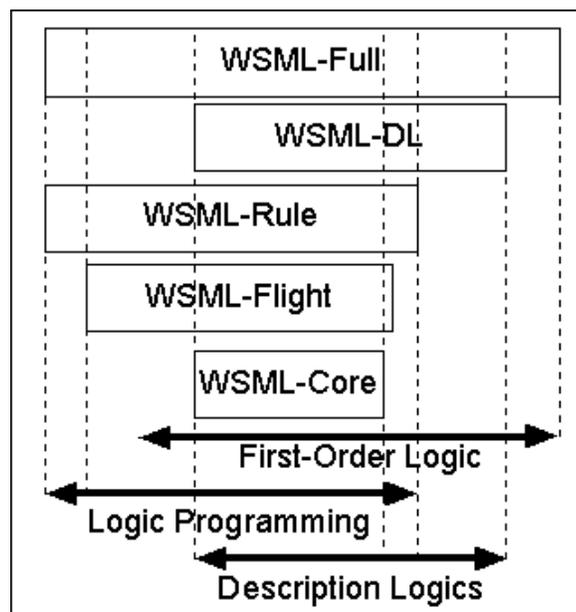


Figure 2. WSML Layering

As can be seen from [Figure 2](#), WSML has two alternative layerings, namely, WSML-Core \rightarrow WSML-DL \rightarrow WSML-Full and WSML-Core \rightarrow WSML-Flight \rightarrow WSML-Rule \rightarrow 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 one of the two paradigms, thereby effectively adding the expressiveness of one of the two paradigms to the intersection (cf. [Levy

[&Rousset, 1998\]](#)) or by reducing the interface between the two paradigms and reason with both paradigms independently (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 [\[SableCC\]](#). 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-terminals. 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.

Structure of the Document

The remainder of this document is structured as follows:

[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. Finally, [Chapter 7](#), 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.

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 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 C contains a complete list of WSML keywords, as well as references to the sections in the document where these are described. Finally, Appendix D contains the changelog.

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
           | capability
           | interface

```

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 belong to the WSML prologue and 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

A WSML specification contains all information about a class and its attributes, a relation and its parameters and an instance and its attribute values 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 WSML, 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. In case one needs to model a property which is independent of the concept definition, this property is most likely a relation rather than an attribute and thus should be

modeled as a relation.

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 into two parts. The first part provides meta-information about the specification, which consists of such things as WSML variant identification, namespace references, annotations, import of ontologies, references to mediators used 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, and finally the use of MIME types for WSML specifications. 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 [[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).

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. Additionally a new kind of identifier has been introduced,

namely variable. It might only be used (a) when using the conceptual syntax in the **nonFunctionalProperty** definition or in the **sharedVariables** block, or (b) when using the logical expression syntax.

```

id          = | iri
              | anonymous
basevalue = datavalue
              | id
idlist      = id
              | '{' id ( ',' id )* '}'
              | '{' '}'

```

Note that the sets of identifiers of the following items are disjoint: ontology, goal, Web service, ooMediator, ggMediator, wgMediator, wwMediator, capability, interface, choreography, orchestration, state signature, grounding identifier, variant identifier, datatype wrapper identifier, built-in predicate identifier.

If 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 in a connective way. For example, if there are two concept definitions which are attached to the same concept identifier, the resulting concept definition includes all attributes of the original definitions. Also if the same attribute is defined in both concept definitions, the range of the resulting attribute will be equivalent to the connection of the ranges of the original attributes.

Data values

WSMML has direct support for different types of concrete data, namely, strings, integers and decimals, which correspond to the XML Schema [Biron & Malhotra, 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](#).

To construct data values, WSMML uses datatype wrappers. *Datatype wrappers* are IRIs used to define data types based on the XML Schema datatypes. Each datatype wrapper has a name and a set of arguments. The name of a datatype wrapper corresponds to the IRI of the datatype. The use of such 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 WSMML as:

`xsd#date(2005,2,3)`.

The arguments of a datatype wrapper can be strings, integers, decimals or variables. No other arguments are allowed for such data terms. Each conforming WSMML implementation is required to support at least the *string*, *integer* and *decimal* datatypes.

The WSMML surface syntax allows, but discourages the use of a shortcut syntax for the datatype wrappers. E.g. `_integer` is short for `xsd#integer` (which is the Compact URI corresponding to the IRI <http://www.w3.org/2001/XMLSchema#integer>). A shortcut syntax is a legacy feature.

Datatype identifiers manifest themselves in WSMML in two distinct ways, namely, as concept identifiers and as datatype wrappers. A datatype wrapper is used as a data structure 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.

Examples of data values:

```
xsd#date(2005,3,12)
xsd#boolean("true")
```

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

```
age ofType xsd#integer
hasChildren ofType xsd#boolean
```

WSMML 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 (`\`) character: `\`.
- Integer values can simply be written as such. Thus an integer of the form *integer* is a shortcut for `xsd#integer("integer")`. For example, `4` is a shortcut for `xsd#integer("4")`.
- Decimal values can simply be written as such, using the `'.'` as the decimal character. Thus a literal of the form *decimal* is a shortcut for `xsd#decimal("decimal")`. For example, `4.2` is a shortcut for `xsd#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 any conforming WSMML application must be able to support, as well as the infix notation which serves as a shortcut for the built-ins.

A data value identifier can be either a simple datavalue, i.e. a number (*integer* or *decimal*) or a *string*, or it can be a constructed datavalue, where a *constructeddatavalue* corresponds to a datatype wrapper.

```

datavalue          =  numberorstring
                    |  constructeddatavalue
numberorstrings    =  numberorstring ( ';' numberorstring )*
numberorstring     =  number
                    |  string
number             =  integer
                    |  decimal
constructeddatavalue = iri '(' numberorstrings ')'

```

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 a Compact URI. Such a Compact URI is similar to the qualified Name 'QName' in XML. After its introduction in XML [[Bray et al., 2004](#)], the term 'QName' has been used with different meanings. The meaning of the term 'QName' as defined in XML 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. WSMML 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 'Compact URIs'.

A Compact URI consists of two parts, namely, the namespace prefix and the local part. A Compact URI is written using a namespace prefix and a localname, separated by a hash ('#'): *namespace_prefix#localname*.

A Compact URI is equivalent to the IRI which is obtained by concatenating the namespace IRI (to which the prefix refers) with the local part of the Compact URI. Therefore, a Compact URI can be seen as an abbreviation for an IRI which enhances the legibility of the specification. If a Compact URI has no prefix, the namespace of the Compact URI is the default namespace of the document. *Note:* In case the default namespace is not defined or a prefix used in a

Compact URI cannot be resolved, the WSMML specification is not well formed.

IRIs are Unicode strings which are valid absolute IRIs according to [Duerst & Suignard, 2005] delimited with the symbols ' _ " ' and ' " '. For convenience, a Compact URI does not require special delimiters. Any Compact URI contains a namespace prefix. Namespace prefixes may not coincide with any WSMML keywords. The characters '.' and '-' in a Compact URI need to be escaped using the backslash ('\') character.

```

full_iri      = ' _ " ' iri_f "'
compactURI = (name '#')? name
iri          = full_iri
              | compactURI

```

Examples of full IRIs in WSMML:

```

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

```

Examples of Compact URIs in WSMML (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>)

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.

The use of an identifier in the specification of the WSMML elements *ontology*, *goal*, *Web service*, *mediator*, *capability*, *interface*, *choreography* and *orchestration* is optional. If no identifier is specified and if the element is not contained within another element (as e.g. a capability contained in a goal), the following rule applies: the identifier is assumed to be the same as the locator of the specification, i.e. the location where the specification was found. If an element is contained within another element, it does not need to have an identifier. The fact that the set of identifiers of those items need to be disjoint, as described in the beginning of this section, makes that there may not be two such top level elements without an identifier in a single file (e.g. a Web service and an ontology without an identifier), as they would otherwise get the same identifier.

Anonymous Identifiers

An anonymous identifier represents an 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 unnumbered anonymous IDs. 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.

Please note that anonymous identifiers are disallowed for the following elements:

- the top-level elements *ontology*, *goal*, *webService*, *ooMediator*, *ggMediator*, *wgMediator* and *wwMediator*.
- *capability*, *interface*, *choreography* and *orchestration*.

Variables

Variables are a new kind of identifiers which are used in defining WSMML logical expressions (see Section [Section 2.8](#)).

Variable names start with an initial question mark, "?". Variables may only occur in place of concepts, attributes, instances, relation arguments or attribute values. A variable may not, for example, replace a WSMML keyword however.

Variables may only be used (a) inside of **logical expressions**, and (b) as values in a **non-functional property** definition (see Section [Section 2.4.3](#)) or in the **sharedVariables** block within capability definitions (see Section [Section 2.4.1](#)).

The scope of a variable is 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

Invalidity of identifiers

Note that whenever in place of an identifier a symbol is used which is not a valid identifier, the document is not a valid WSMML document. Examples include Compact URIs with undefined namespace prefixes and invalid data values.

2.1.3 Comments in WSMML

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

comment = short_comment | long_comment
short_comment = '/' ' not_cr_lf'* eol
long_comment = '/' '*' long_comment_content* '*' '/'

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

Examples:

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

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

2.1.4 WSML MIME type

When accessing resources over the Web, the MIME type indicates the type of the resource. For example, plain text documents have the MIME type `text/plain` and XML documents have the MIME type `application/xml`.

When exchanging WSML documents written in the normative syntax, it is necessary to use the appropriate MIME type so that automated agents can detect the type of content. The MIME type to be used for WSML documents is: **`application/x-wsml`**

WSML/XML documents should have the MIME type: **`application/x-wsml+xml`**
 WSML/RDF documents in the XML serialization of RDF should have the usual MIME type: **`application/rdf+xml`**

2.2 WSML Prologue

This section describes the prologue of a WSML specification, that is all those elements that are 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 `wsmIVariant` 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 `wsmIVariant` is optional. In case no variant is specified, no

guarantees can be made with respect to the specification and WSMML-Full may be assumed.

wsmlvariant = 'wsmlVariant' full_iri

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

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

When the intended WSMML variant is explicitly stated, tools can immediately recognize the intention of the author and return an exception if the specification does not conform to the syntactical restrictions imposed by the intended variant. This generally helps developers of WSMML 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 WSMML document, below the identification of the WSMML 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. Note that, like any argument list in WSMML, 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 are given below, 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/",
  wsmml _"http://www.wsmo.org/wsmml-syntax#",
  loc _"http://www.wsmo.org/ontologies/location#",
  oo _"http://example.org/ooMediator#"}
```

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

2.2.3 Header

The WSMML header consists of items that any WSMML specification may have: annotations, import ontologies and use mediators:

```
header = annotations
          | importsontology
          | usesmediator
```

Annotations

Annotations can be used for all the WSMML elements. Annotation blocks are delimited with the keywords **annotations** and **endAnnotations**. Following the keyword is a list of attribute values, which consists of the attribute identifier, the keyword **hasValue** and the value for the attribute. This value may be any identifier and can thus be an IRI, a data value, an unnumbered anonymous identifier or a comma-separated list of the former, delimited with curly brackets. The recommended properties are the properties of the Dublin Core Metadata Element Set [Weibel et al. 1998], but the list of properties is extensible and the user can thus choose to use properties coming from different sources. WSMO [Roman et al., 2006] defines a number of properties which do not belong to Dublin Core; they can be used in a WSMML specification by referring to the WSMML namespace (<http://www.wsmo.org/wsmml/wsmml-syntax#>). These properties are: **wsmml#version** and **wsmml#owner**, (here we assume that the prefix *wsmml* has been defined as referring to the WSMML namespace; see Section 2.1.1). For the recommended usage of annotations see [Roman et al., 2006].

```
annotations      = 'annotations' annotationvalue* 'endAnnotations'
annotationvalue = id 'hasValue' basevaluelist
```

Example:

```
annotations
dc#title hasValue "WSMML example ontology"
dc#subject hasValue "family"
dc#description hasValue "fragments of a family ontology to provide WSMML 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 xsd#date("2004-11-22")
dc#format hasValue "text/html"
dc#language hasValue "en-US"
dc#rights hasValue "_http://www.deri.org/privacy.html"
wsmml#version hasValue "$Revision: 1.238 $"
endAnnotations
```

Annotations in WSMML are not part of the logical language, that is of the conceptual or the logical expression syntax in WSMML; programmatic access to these properties can be provided through an API.

The current version of WSMML is not backward compatible with respect to the syntax used to specify annotations descriptions. In the previous versions (versions before [d16.1v0.3_20061110](#)) annotation descriptions were implemented as *nonFunctionalProperties* or *nfp* constructs. The current versions use the *annotations* construct. A translation tool from the old syntax to the new syntax to convert documents is available at <http://tools.deri.org/wsml/nfptranslator/>.

Importing Ontologies

Ontologies may be imported in any WSMML 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, these ontologies are also imported.

importsontology = 'importsOntology' irilist

Example:

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

WSMML does not only allow the import of WSMML ontologies: the imported ontologies may be WSMML ontologies, RDF graphs, or OWL DL ontologies. The variant of every imported ontology should be lower than or equal to the variant of the importing description. RDFS ontologies are seen as having the variant WSMML Flight. If an OWL DL ontology is in the DLP subset, i.e., it is equivalent to a set of equality-free Horn formulas, it is seen as having the WSMML-Core variant. If not, it is seen as having the variant WSMML-DL. See [[de Bruijn, 2008b, Section 3.1.1](#)] for more details.

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

When importing RDFS or OWL DL ontologies into WSMML, it is possible to refer to elements in these ontologies from WSMML formulas. RDFS/OWL classes and properties correspond to WSMML concepts and attributes, respectively. The following table shows the basic statements in RDF and OWL DL and the corresponding statements in the WSMML logical expression syntax.

RDFS/OWL construct	Example	WSML construct	Example
RDF Class membership	a rdf:type C	Concept instance	a memberOf C
RDF Property value	a p b	Attribute value	a[p hasValue b]
RDFS Subclass	C rdfs:subClassOf D	Subconcept statement	C subConceptOf D
OWL DL Class membership	Individual(a type(C))	Concept instance	a memberOf C
OWL DL Property value	Individual(a value(p b))	Attribute value	a[p hasValue b]
OWL DL Subclass	SubClass(C D)	Subconcept statement	C subConceptOf D

Using Mediators

Mediators are used to link different WSML elements (ontologies, goal, web services) and resolve heterogeneity between the elements. Mediators are described in more detail in [Section 2.5](#). Here we are 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' irilist

Example:

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

2.3 Ontologies 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. If no identifier is specified for the ontology, the locator of the ontology serves as identifier.

Example:

```
ontology family
```

An ontology specification document in WSML consists of:

ontology = 'ontology' iri? header* ontology_element*

```

ontology_element =  concept
                   | relation
                   | instance
                   | relationinstance
                   | axiom

```

In this section we explain the ontology modeling elements in the WSMML language. The modeling elements are based on the WSMO conceptual model of ontologies [Roman et al., 2006].

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 **annotations** block and zero or more attribute definitions.

Note that WSMML allows inheritance of attribute definitions, which means that a concept inherits all attribute definitions of its superconcepts. If two superconcepts have a 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? annotations? attribute*
superconcept    = 'subConceptOf' idlist

```

Example:

```

concept Human subConceptOf {Primate, LegalAgent}
annotations
  dc:description hasValue "concept of a human being"
  dc:relation hasValue humanDefinition
endAnnotations
hasName ofType foaf#name
hasRelative impliesType Human
hasAncestor impliesType Human
hasParent impliesType Human
hasChild impliesType Human
hasWeightInKG ofType xsd#float
hasBirthdate ofType xsd#date
dateOfDeath ofType xsd#date
hasBirthplace ofType loc#location
isMarriedTo impliesType Human
hasCitizenship ofType oo#country

```

WSMML allows creation of 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 WSMML specification to include the relation between the concept and the axioms related to the concept in the annotations 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. 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. WSMML supports defined concepts through the use of axioms (see Section 2.3.4). The axiom definition below specifies that each instance of `Human` is also an instance of `Primate` and `LegalAgent`. Furthermore, all values for the attributes `hasName`, `hasParent`, `hasWeightInKG` 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 a member of this concept.

As mentioned above axioms can be used to define concepts. 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:

$$?x \text{ memberOf } A \text{ equivalent } ?x \text{ memberOf } B_1 \text{ and } \dots \text{ and } ?x \text{ memberOf } B_n$$

With A and B_1, \dots, 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

As already described in [Section 2.1, WSMML Syntax Basics](#), attributes in WSMML 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. In case one needs to model a property which is independent of the concept definition, this property is most likely a relation rather than an attribute and thus should be modeled as a relation.

WSMML allows two kinds of attribute definitions, namely, constraining definitions with the keyword **ofType** and inferring definitions with the keyword

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

An attribute definition using the keyword **ofType**, 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 if 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.

It can sometimes be useful to use both **ofType** and **impliesType** for the same attribute. This is achieved by including two attribute definitions for the attribute. For example, `hasMother` could have its range restricted to `FemalePerson` but conclude that any filler is a `Mother`.

Please note that WSMML allows instances which are not members of a particular concept. This makes that the attribute range does not need to contain a concept identifier or datatype as value. Attributes can thus be written in all WSMML variants by using an empty idlist, e.g. ' a **ofType** { }'.

Attributes which do not have a datatype 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. Attributes can also be specified as being a subAttribute of another attribute, using the keyword **subAttributeOf**. Notice that these keywords do not enforce a constraint on the attribute, but are used to infer additional information about the attribute. The keywords **inverseOf** and **subAttributeOf** must be followed by an identifier of the attribute, enclosed in parentheses, of which this attribute is the inverse, respectively the subAttribute of.

WSMML allows the specification of cardinality and range constraints which are defined like integrity constraints in databases, i.e., in the case of violation of a constraint, a given ontology is inconsistent. This additional feature allows to check the integrity of a closed set of data, implicitly leading the modeler to express a local closed world view on her/his published data.

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 *). Note that a maximal cardinality of 1 makes an attribute functional.

```

attribute          = id attributefeature* att_type cardinality? idlist
                       annotations?
att_type           = 'ofType'
                       | 'impliesType'
cardinality        = '(' digit+ cardinality_number? ')'
cardinality_number = digit+
                       | '*'
attributefeature   = 'transitive'
                       | 'symmetric'
                       | 'inverseOf' '(' id ')'
                       | 'subAttributeOf' '(' 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*. In the same time *att2* is the inverse of *att1* which means that the following can be inferred: *a* has an attribute *att1* with value *b*. The *inverseOf* relationship is a symmetric relationship between attributes. This means that if *att1* is the inverse of *att2*, then *att2* is the inverse of *att1*.

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

Below is an example of a concept definition with attribute definitions:

```

concept Human
  annotations
    dc:description hasValue "concept of a human being"

```

endAnnotations

hasName **ofType** foaf#name
 hasRelative **symmetric impliesType** Human
 hasAncestor **transitive impliesType** Human
 hasParent **inverseOf**(hasChild) **subAttributeOf**(hasAncestor) **impliesType** Human
 hasMother **ofType** FemaleHuman
 hasMother **impliesType** Mother
 hasChild **subAttributeOf**(hasRelative) **impliesType** Human
 hasWeightInKG **ofType** (1) xsd#float
 hasBirthdate **ofType** (1) xsd#date
 dateOfDeath **ofType** (0 1) xsd#date
 hasBirthplace **ofType** (1) loc#location
 isMarriedTo **symmetric impliesType** (0 1) Human
 hasCitizenship **ofType** oo#country

2.3.2 Instances

An instance definition starts with the **instance** keyword, optionally followed by the identifier of the instance and optionally by 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. Attribute values can be IRIs, unnumbered anonymous identifiers and datavalues.

instance = 'instance' id? memberOf? annotations? attributevalue*
memberOf = 'memberOf' idlist
attributevalue = id 'hasValue' valuelist

Example:

```

instance Mary memberOf {Parent, Woman}
annotations
  dc#description hasValue "Mary is parent of the twins Paul and Susan"
endAnnotations
  hasName hasValue "Maria Smith"
  hasBirthdate hasValue xsd#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., 2006], is achieved 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.

WSML allows instances which are not members of a particular concept. This can

be written in all WSMML variants like, e.g.

```
instance Mary
  hasName hasValue "Maria Smith"
```

2.3.3 Relations

Relations in WSMML can be used in order to model interdependencies between several concepts (respectively instances of these concepts). A relation definition starts with the **relation** keyword, which is optionally followed by the identifier of the relation. WSMML 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 optionally completed by the keyword **subRelationOf** followed by one or more identifiers of superrelations. Finally an optional **annotations** block can be specified.

Relations in WSMML 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 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? paramdef superrelation? annotations?
paramdef     = arity
                | paramtyping
arity        = '/' pos_integer
paramtyping  = '(' paramtype moreparamtype* ')'
paramtype    = att_type idlist
moreparamtype = ',' paramtype
superrelation = 'subRelationOf' idlist
```

Below are two examples, one with parameter definitions and one with an arity definition:

```
relation distanceInKm (ofType City, ofType City, impliesType _decimal) subRelationOf measurement
```

```
relation distanceInKm/3
```

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 that related axioms are indicated using the annotation `dc#relation`.

2.3.4 RelationInstances

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, and the name of the relation to which the instance belongs. This is followed by the values of the parameters associated with the instance and by an optional **annotations** block. The parameter values can be IRIs, unnumbered anonymous identifiers and datavalues.

```
relationinstance = 'relationInstance' [name]: id? [relation]: id '(' value
morevalues ')' annotations?
```

Below is 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)
```

2.3.5 Axioms

The usage of axioms in WSMML allows for example to refine the definition already given in the conceptual syntax. This can concern for example subconcept and attribute definitions (see also [Section 2.3.1](#)).

An axiom definition starts with the **axiom** keyword, optionally followed by the name (identifier) of the axiom. This is followed by an optional **annotations** block and one or more logical expression(s) preceded by the **definedBy** keyword. The logical expressions must be followed by either a blank or a new line. The language allowed for the logical expressions is explained in [Section 2.8](#).

```
axiom = 'axiom' axiomdefinition
axiomdefinition = id? annotations?
| id? annotations? log_definition
log_definition = 'definedBy' log_expr+
```

Example of an axiom definition:

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

WSMML allows the specification of database-style constraints. Below is 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 Capabilities, Interfaces and Non-functional Properties in WSML

The desired and provided functionality of services 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 is described in interfaces, as part of the goal and the Web service, respectively. Finally, other aspects of services or goals which are neither functional nor behavioral are captured in non-functional properties.

2.4.1 Capabilities

A capability constitutes a formal description of the functionality requested from or provided by a Web service. WSML supports two ways of defining capabilities: create a complex capability description consisting of annotations, imported ontologies, used mediators, non-functional properties, shared variables, assumptions, preconditions, effects and postconditions, or refer to one concept (using an IRI) which is presumably defined in an ontology imported by the goal or Web service.

Concerning the complex capability descriptions, the preconditions describe conditions on the input of the service, the postconditions describe the relation between the input and the output of the service, the assumptions describe what must hold (but cannot be checked beforehand) of the state of the world for the Web service to be able to execute successfully, and the effects describe real-world effects of the execution of the Web service which are not reflected in the output.

A WSML goal or Web service may only have one capability. The specification of a capability is optional. Please note that all ontologies that are imported within a goal or Web service specification are implicitly imported in all the contained (complex) capabilities as well.

A WSML capability may be a stand-alone entity which may be defined outside a WSML goal or Web service definition. If a WSML capability is a stand-alone entity then it cannot be written in the same file as goal, Web service and mediator descriptions. A capability written in a file underneath a goal or Web service belongs to that goal or Web service. A capability that is written in a file in a way that makes it impossible to see to which Web service or goal it belongs to, causes a parse error.

A complex capability description starts with the **capability** keyword, (optionally) followed by the name (identifier) of the capability. This is followed by an optional

annotations block, an optional **importsOntology** block, an optional **usesMediator** block and one or more optional **nonFunctionalProperty** statements. The (optional) **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 **annotations** 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 WSMML variant and is explained in the respective chapters.

A simple capability description starts with the **capability** keyword, followed by an IRI depicting a concept within an external ontology, describing the functionality of the goal or Web service. Such a capability should not define any annotations; they should be written in the referred ontology.

capability = 'capability' iri? header* nfp* sharedvardef?
pre_post_ass_or_eff*

sharedvardef = 'sharedVariables' variablelist

pre_post_ass_or_eff = 'precondition' axiomdefinition
| 'postcondition' axiomdefinition
| 'assumption' axiomdefinition
| 'effect' axiomdefinition

Below is an example of a capability specified in WSMML:

```

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

  effect
    annotations
      dc:description hasValue "After the registration the child
        is a German citizen"
    endAnnotations
    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. Please note that all ontologies that are imported within a goal or Web service specification are implicitly imported in all the contained capabilities as well.

A WSML interface may be a stand-alone entity which may be defined outside a WSML goal or Web service definition. If a WSML interface is a stand-alone entity then it cannot be written in the same file as goal, Web service and mediator descriptions. An interface written in a file underneath a goal or Web service belongs to that goal or Web service. An interface that is written in a file in a way that makes it impossible to see to which Web service or goal it belongs to, causes a parse error.

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

Currently only a syntax for choreographies is defined in WSML. A choreography specification starts with the **choreography** keyword, (optionally) followed by the name (identifier) of the choreography. This is followed by an optional **annotations** block, an optional **importsOntology** block and an optional **usesMediator** block. Following these elements are the optional blocks of **stateSignature** and **transitions** containers. An orchestration simply starts with the **orchestration** keyword followed by an optional identifier. This element is yet to be better defined.

```

interface      = 'interface' iri? header* nfp* choreography? orchestration?
choreography   = 'choreography' iri? header* state_signature? transitions?
orchestration  = 'orchestration' iri?

```

A state signature in a choreography description starts with the **stateSignature** keyword followed by an optional identifier, an optional block of **annotations**, an optional **importsOntology** statement and an optional **usesMediator** statement. Finally, the state signature defines an optional set of mode containers. Each mode container is defined by a mandatory mode name (**static**, **in**, **out**, **shared**, **static** or **controlled**) and a list of entries. Each entry may take a default form (which relates to a concept), an explicit concept definition or an explicit relation definition. The default form is defined by an IRI followed by an optional block of grounding IRIs. An explicit concept mode entry is defined by the keyword **concept** followed by an IRI followed by an optional block of grounding IRIs. The same applies for a relation mode entry except that instead of the keyword **concept**, **relation** is used . The

grounding information is defined by the **withGrounding** keyword followed by a list of IRIs. Note that we refrain from defining the state since such an element is comprised of the ground facts defined in the imported ontologies.

<u>statesignature</u>	= 'stateSignature' <u>id?</u> <u>header*</u> <u>mode*</u>
<u>mode</u>	= <u>grounded_mode</u> <u>ungrounded_mode</u>
<u>grounded_mode</u>	= 'in' <u>grounded_mode_entry_list</u> 'out' <u>grounded_mode_entry_list</u> 'shared' <u>grounded_mode_entry_list</u>
<u>ungrounded_mode</u>	= 'static' <u>ungrounded_mode_entry_list</u> 'controlled' <u>ungrounded_mode_entry_list</u>
<u>grounded_mode_entry_list</u>	= <u>grounded_mode_entry</u> <u>grounded_mode_entry</u> , <u>grounded_mode_entry_list</u>
<u>ungrounded_mode_entry_list</u>	= <u>ungrounded_mode_entry</u> <u>ungrounded_mode_entry</u> , <u>ungrounded_mode_entry_list</u>
<u>grounded_mode_entry</u>	= <u>iri grounding</u> 'concept' <u>iri grounding</u> 'relation' <u>iri grounding</u>
<u>ungrounded_mode_entry</u>	= <u>iri</u> 'concept' <u>iri</u> 'relation' <u>iri</u>
<u>grounding</u>	= 'withgrounding' <u>irilist</u>

Following the state signature block is the transition rules container. This is defined by the **transitionRules** keyword followed by an optional identifier, an optional **annotations** block and a set of rule elements. A rule can take the form of an if-then, a choose, a for-all or an update-rule. An if-then rule is defined by the **if** keyword, a WSML Logical Expression (condition), a **then** keyword, a non-empty set of rule elements and the **endif** keyword. A for-all rule is defined by the **forall** keyword, a list of variable elements, a **with** keyword, a WSML Logical Expression (condition), a **do** keyword, a set of non-empty rule elements and the **endforall** keyword. Similarly, a choose rule is defined by the **choose** keyword, a list of variable elements, a **with** keyword, a WSML Logical Expression (condition), a **do** keyword, a set of non-empty rule elements and the **endchoose** keyword. Each of the if-then, choose and for-all rule is followed by an (optional) piped rule using the operator "|", meaning that one of the rules is chosen non-deterministically.

transitions = 'transitionRules' id? annotations? rule*

```

rule      = 'if' condition 'then' rule+ 'endif' pipedReaderRule?
            | 'forall' variablelist 'with' condition 'do' rule+
              'endforall' pipedReaderRule?
            | 'choose' variablelist 'with' condition 'do' rule+ | updateRule
              'endChoose' pipedReaderRule?

condition = expr
pipedReaderRule = '|' rule

```

An update rule is defined by a modifier keyword (**add**, **delete** or **update**), an open parenthesis, a fact and a closing parenthesis. A fact can take the form of a preferred molecule, a non-preferred molecule or a fact molecule. The first form, the preferred molecule, is defined by a term, an optional attribute fact, a **memberOf** keyword, a term list and an optional fact update. An attribute fact is defined by an open square bracket, an attribute fact list and a closing square bracket. An attribute fact list can take the form of an attribute relation or a list of attribute relations (defined recursively) delimited by a comma. An attribute relation is defined by a term, a **hasValue** keyword, a term list and an optional fact update. A fact update is defined by an arrow of the form => followed by a term list. The non-preferred form of a fact is defined by a term, a **memberOf** keyword, a term list, an optional fact update and an attribute fact. A fact molecule is defined by a term and an attribute fact. A fact can also be related to a relation. Such a fact is defined by the '@' symbol (which clearly identifies that the modifier deals with an update of a relation) an identifier, an open parenthesis, a list of term updates (depending on the number of arguments of the relation) and a closing parenthesis. The list of term updates is defined as a single term update or a term update followed by a comma followed by another term update. A term update is defined as a term, or as a term followed by the update symbol => and another new term.

```

updateRule   = modifier '('fact')'
modifier     = 'add'
              | 'delete'
              | 'update'

fact         = term attr fact? 'memberOf' termlist fact update?
              | term 'memberOf' termlist fact update? attr fact
              | term attr fact
              | '@' id '('term updates')'

fact update  = '=>' termlist
attr fact    = '[' attr fact list']'
attr fact list = term 'hasValue' termlist fact update?
                  | attr fact list',' term 'hasValue' termlist fact update?

term updates = term update
                | term update',' term updates

```

```

term_update = term
                | [oldterm]: term '=>' [newterm]: term
new_term    = '=>' term

```

The syntax presented before, known more commonly as the WSMML Abstract State Machine (ASM) syntax, is the only syntax for choreographies that may be embedded within a WSMML description. Should other formalisms and syntaxes be desired these choreographies should be referenced by IRIs. As no orchestration syntax has so far been specified by the WSMML group it is currently not possible to embed an orchestration within a WSMML description, however, same as choreographies, they may be referenced by IRIs.

Below is an example of an interface, an example of references to multiple interfaces and an example of an interface with a defined choreography:

```

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

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

interface buyInterface

  choreography buyChoreography
  annotations
    dc#title hasValue "Multimedia Shopping Service Choreography"
    dc#description hasValue "Describes the steps required for shopping multimedia items over this web service"
  endAnnotations

  stateSignature

  importsOntology {
    _"http://example.org/ontologies/products/Products",
    _"http://example.org/ontologies/tasks/ShoppingTasks",
    _"http://example.org/ontologies/shopping/Shopping",
    _"http://example.org/ontologies/products/MediaProducts",
    _"http://example.org/ontologies/Media"
  }

  in
    shoptasks#SearchCatalog withGrounding
    _"http://example.org/webservices/shopping/mediashoppingservice#wsdl.interfaceMessageReference(MediaShopping"

  out
    mediaproduct#MediaProduct withGrounding
    _"http://example.org/webservices/shopping/mediashoppingservice#wsdl.interfaceMessageReference(MediaShopping"

  transitionRules

  forall ?search
    with
      (?search[

```

```

    byTitle hasValue ?title,
    byArtist hasValue ?artist,
    byMinPrice hasValue ?minPrice,
    byMaxPrice hasValue ?maxPrice,
    byMinRating hasValue ?minRating,
    byMaxRating hasValue ?maxRating
] memberOf shoptasks#SearchCatalog
and ?artist memberOf media#Artist
and exists?item(
    ?item memberOf mediaproduct#MediaProduct and(
    ?item[hasContributor hasValue ?artist] or
    ?item[hasTitle hasValue ?title] or
    (
        ?item[hasPrice hasValue ?price] and
        ?price >= ?minPrice and
        ?price <= ?maxPrice
    ) or
    (
        ?item[hasRating hasValue ?rating] and
        ?rating >= ?minRating and
        ?rating <= ?maxRating
    )
    )
)
)
)
do
add(?item[
    hasContributor hasValue ?artist,
    hasTitle hasValue ?title,
    hasPrice hasValue ?price,
    hasRating hasValue ?rating
] memberOf mediaproduct#MediaProduct
)
add(?artist memberOf media#Artist)
delete(?search memberOf shoptasks#SearchCatalog)
endForall

```

2.4.3 Non-functional Properties

Non-functional properties are those properties which strictly belong to a Web service, goal, capability, interface or mediator and which are not functional and behavioral. A WSMML Web service, goal, capability, interface or mediator may specify multiple non-functional properties. The specification of a non-functional property is optional.

Non-functional properties start with the keyword **nonFunctionalProperty** or its short form **nfp**. Following the keyword is an attribute value, which consists of the attribute identifier, the keyword **hasValue** and the value for the attribute, which may be an identifier, a data value or a variable symbol (or a set of such symbols). This is followed by an optional **annotations** block.

Note that only one attribute value per non-functional property can be defined.

In case the value of the attribute in the name-value pair is an identifier, this

value may be an IRI, a data value, an anonymous identifier or a comma-separated list of the former, delimited with curly brackets. In case it is a variable symbol, an axiom definition block in which the variable symbol is used should follow. Such an axiom definition block consists of the keyword **definedBy**, followed by one or more logical expressions, each followed by either a blank or a new line character (please note that an axiom definition block as defined for non-functional properties is not equal to an axiom as defined in Section [Section 2.3.4.](#) The keyword **axiom** is omitted here.). The language allowed for the logical expression differs per WSMML variant. The logical expressions are restricted to rule bodies for the Core, Flight and Rule variants, and to descriptions (i.e. tree-shaped formulas) for the DL variant. If the axiom definition block is missing or the variable symbol is not used in it, then implementations should issue a warning to the user.

```

nfp = 'nfp' attributevaluenfp log_definition_nfp?
      | 'nonFunctionalProperty' attributevaluenfp log_definition_nfp?
attributevaluenfp = id 'hasValue' valuelistnfp annotations?
valuenfp          = basevalue
                    | variable
valuelistnfp     = valuenfp
                    | '{' valuenfp morenfpvalues* '}'
morenfpvalues   = ',' valuenfp
log_definition_nfp = 'definedby' log_expr+

```

Below is an example of non-functional properties specified in WSMML:

```

nonFunctionalProperty
  po#Price hasValue ?price
  annotations
    dc#description hasValue "If the client is older than 60 or younger than 10 years old the
    invocation price is lower than 10 euro"
  endAnnotations
  definedBy
    ?client[age hasValue ?age] memberOf hu#human and ?age[amount hasValue ?years, units hasValue hu#Year]
    memberOf hu#age and (greaterEqual(?years, 60) or lessEqual(?years,10))
    implies ?price[hasAmount hasValue ?amount, hasCurrency hasValue cur#Euro] memberOf
    po#AbsoultePrice and lessEqual(?amount, 10).

```

2.5 Goals in WSMML

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

Example:

```

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

```

A goal specification document in WSML consists of:

```
goal = 'goal' iri? header * nfp * capability ? interface *
```

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

Please note that all ontologies that are imported within a goal specification are implicitly imported in all the contained capabilities and interfaces as well.

2.6 Web Services 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' iri ? header * nfp * capability ? interface *
```

Example:

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

The elements of a Web service are capability and interface which are explained in [Section 2.4](#).

Please note that all ontologies that are imported within a Web service specification are implicitly imported in all the contained capabilities and interfaces as well.

2.7 Mediators 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. [\[Roman et al., 2006\]](#)).

```
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.

Example:

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

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

```
use_service = 'usesService' iri
source      = 'source' iri
msources   = 'source' '{' iri moreiris* '}'
sources    = source
              | msources
target     = 'target' iri
```

2.7.1 ooMediators

ooMediators are used to connect ontologies to other ontologies, Web services, goals and mediators. ooMediators take care of resolving any heterogeneity which occurs.

The **source** of an ooMediator in WSMML may only contain identifiers of ontologies and other ooMediators as source. ooMediators can have more than one source.

An ooMediator in WSMML 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 iri.

```
oomeediator = 'ooMediator' iri? baseheader* nfp* sources? target?
              use_service?
baseheader = annotations
              | importsontology
```

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 WSMML ontology into another WSMML 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, usually 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.

2.7.2 wwMediators

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

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

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

wwmediator = 'wwMediator' iri? header* nfp* source? target? use_service?

2.7.3 ggMediators

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

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

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

ggmediator = 'ggMediator' iri? header* nfp* source? target? use_service?

2.7.4 wgMediators

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

wgMediators in WSML may have multiple **sources**. A source may be an identifier of a Web service or another wgMediator.

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

wgmediator = 'wgMediator' iri? header* nfp* sources? 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 [\[Roman et al., 2006\]](#).

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 [\[Keller et al., 2005\]](#), these descriptions often need a very expressive language.

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 imposes on the logical expression syntax.

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 $?alphanum^*$.

Object constructors are those elements of V that can be either functional symbols, datatype wrappers, data values, or anonymous identifiers.

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. WSML also allows the use of **true** and **false** as predicates with the standard pre-defined meaning.

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, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.
- If $f \in V_D$ and dv_1, \dots, dv_n are in $V_{DV} \cup V_V$, then $f(dv_1, \dots, dv_n)$ 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. A logical expression is a formula 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 $\alpha, \beta \in Term(V)$ and $\gamma \in Term(V)$ or γ is of the form $\{ \gamma_1, \dots, \gamma_n \}$, with $\gamma_1, \dots, \gamma_n \in Term(V)$, then:
 - α **subConceptOf** γ is an atomic formula in $L(V)$. Here, α and γ both identify concepts.
 - α **memberOf** γ is an atomic formula in $L(V)$. Here, α identifies an instance and gamma identifies (a) concept(s).
 - $\alpha[\beta$ **ofType** $\gamma]$ is an atomic formula in $L(V)$. Here, α identifies a concept, β identifies an attribute and γ identifies (a) concept(s).
 - $\alpha[\beta$ **impliesType** $\gamma]$ is an atomic formula in $L(V)$. Here, α identifies a concept, β identifies an attribute and γ identifies (a) concept(s).
 - $\alpha[\beta$ **hasValue** $\gamma]$ is an atomic formula in $L(V)$. Here, α identifies an instance, β identifies an attribute and γ identifies (an) instance(s).

These atomic formulae are also called *molecules*. Notice that an identifier can play a different role (instance, attribute, concept), depending on the use in a molecule.
- If $r \in V_R$ and t_1, \dots, t_n are terms, then $r(t_1, \dots, t_n)$ is an atomic formula in $L(V)$.
- If $\alpha, \beta \in Term(V)$ then $\alpha = \beta$ and $\alpha \neq \beta$ are atomic formulae 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 $?x_1, \dots, ?x_n$ 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** $?x_1, \dots, ?x_n (\alpha)$ is a formula in $L(V)$.
 - **exists** $?x_1, \dots, ?x_n (\alpha)$ 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 \rightarrow , \leftarrow and \leftrightarrow 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.

To enhance the readability of logical expressions it is possible to abbreviate a conjunction of several molecules 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].
```

WSML keywords are allowed in the WSML logical expression syntax. Additionally, WSML allows the keywords 'true' and 'false' in the human-readable syntax.

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 [Grosz 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 [Grosz 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 WSML-Core Syntax Basics

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 syntax basics.

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_{ANN} are the sets of concept, datatype, relation, instance and annotation 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.

The arguments of a datatype wrapper in WSML-Core can only be strings, integers

or decimals. No other arguments, also not variables, are allowed for such data terms.

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. [\[Grosf et al., 2003\]](#).

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**, **inverseOf** and **subAttributeOf**. This restriction stems from the fact that reflexivity, transitivity, symmetricity and inverse 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 inversivity 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.

One may not specify constraining attribute definitions, other than for datatype ranges. In other words, attribute definitions of the form: $A \text{ ofType } D$ are not allowed, unless D is a datatype identifier.

3.2.2 Instances

In WSML-Core, allowed attribute values are restricted to strings, numbers, ids, or a lists of such values. Constructed data values are not allowed.

3.2.3 Relations

WSML-Core does not allow for the specification of relations.

3.2.4 RelationInstances

WSML-Core does not allow the specification of relation instances, as the use of relations is disallowed.

3.2.5 Axioms

WSML-Core does not impose restrictions on the specification of axioms, apart from 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 'assumptions', 'preconditions', 'effects' and 'postconditions' of a capability and 'definition' of a non-functional property are limited to WSML-Core logical expressions.

3.4. 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' of a capability and 'definition' of a non-functional property are limited to WSML-Core logical expressions.

3.5. Mediators in WSML-Core

Mediators in WSML-Core follow the common WSML syntax.

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.

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 , $\phi \in V_I$, 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 $L(V)$
- γ [s **ofType** Δ] is an atomic formula in $L(V)$
- γ [s **impliesType** Δ] is an atomic formula in $L(V)$
- γ [p **impliesType** Γ] is an atomic formula in $L(V)$
- ϕ [p **hasValue** ψ] is an atomic formula in $L(V)$

- $\phi[s \text{ hasValue } Val]$ is an atomic formula in $L(V)$

Let Var_1, Var_2, \dots be arbitrary WSML variables. We call molecules of the form Var_i **memberOf** Γ *a-molecules*, and molecules of the forms, $Var_i[p \text{ hasValue } Var_k]$ and $Var_i[p \text{ hasValue } \{Var_{k1}, Var_{k2}\}]$ *b-molecules*, respectively.

In the following, F stands for an lhs-formula (i.e., a formula allowed in the antecedent, or *left-hand side*, of an implication), with the set of lhs-formulae defined 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

In the following, G, H stand for rhs-formulae (i.e., formulae allowed in the consequent, or *right-hand side*, of an implication), with the set of rhs-formulae defined as follows:

- Any a-molecule is an rhs-formula
- if G and H are rhs-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 $L(V)$.
- If F_1, \dots, F_n are atomic formulae, then F_1 **and** \dots **and** F_n is a formula in $L(V)$.
- $Var_1[p \text{ hasValue } Var_2]$ **impliedBy** $Var_1[p \text{ hasValue } Var_3]$ **and** $Var_3[p \text{ hasValue } Var_2]$ (globally transitive attribute/relation) is a formula in $L(V)$.
- $Var_1[p \text{ hasValue } Var_2]$ **impliedBy** $Var_2[p \text{ hasValue } Var_1]$ (globally symmetric attribute/relation) is a formula in $L(V)$.
- $Var_1[p \text{ hasValue } Var_2]$ **impliedBy** $Var_1[q \text{ hasValue } Var_2]$ (globally sub-attribute/relation) is a formula in $L(V)$.
- $Var_1[p \text{ hasValue } Var_2]$ **impliedBy** $Var_2[q \text{ hasValue } Var_1]$ (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 (and F **implies** H) 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.
- H **impliedBy** G (and G **implies** H) is a formula in $L(V)$ if all the WSML variables occurring in H occur in G as well.
- Any occurrence of a molecule of the form $Var_1[p \text{ hasValue } Var_2]$ in a WSML-Core clause can be interchanged with $p(Var_1, Var_2)$ (i.e., these two forms can be used interchangeably 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 between Var_1 and Var_2 for every molecule $Var_1[p \text{ hasValue } Var_2]$.

Note that wherever an a-molecule (or b-molecule) is allowed in a WSML-Core

clause, compound molecules abbreviating conjunctions of a-molecules (or b-molecules, respectively), as mentioned in the end of Section 2.8, are also allowed.

The following are examples of WSML-Core logical expressions:

The attribute 'hasAncestor' is transitive:

`?x[hasAncestor hasValue ?z] impliedBy ?x[hasAncestor hasValue ?y] and ?y[hasAncestor hasValue ?z].`

A female person is a woman:

`?x memberOf Woman impliedBy ?x memberOf
Person and ?x memberOf Female`

A student is a person:

`Student subConceptOf Person.`

4. WSML-DL

WSML-DL is an extension of WSML-Core to a full-fledged description logic with an expressiveness similar to OWL DL, namely SHIQ(D). 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 with respect to the WSML-DL semantics also hold with respect to the WSML-Core semantics.

Same as OWL, WSML-DL does not adhere to the Unique Name Assumption (UNA). This means that having two different names does not mean that these refer to two different individuals.

All restrictions on the general conceptual modeling syntax of WSML, introduced in Chapter 2, imposed by WSML-Core, also holds for WSML-DL. The difference between WSML-Core and WSML-DL lies in the logical expression syntax. The logical expression syntax of WSML-DL is less restrictive than the logical expression syntax of WSML-Core. The remainder of this chapter defines the logical expression syntax allowed in WSML-DL.

4.1. WSML-DL Logical Expression Syntax

WSML-DL allows only a restricted form of logical expressions, as an extension of the WSML-Core logical expression syntax. The source of the restrictions on WSML-DL is the fact that the language is restricted to the *SHIQ(D)* subset of First-Order logic [Borgida, 1996]. This subset is close to the two-variable fragment of First-Order Logic (the only description that needs more than two variables is the description of transitive roles); it disallows the use of function symbols, restricts the arity of predicates to unary and binary and prohibits chaining variables over predicates.

Currently the WSML-DL logical expression syntax is specified in first-order logic style. The resulting language is not very user-centric, so that it is hard to use the full expressivity that Description Logics provide to WSML-DL. This is why we will propose a new DL-based alternative, non-normative, surface syntax in the next release of this deliverable.

Definition 4.1. Any WSML-Core vocabulary V is a WSML-DL vocabulary.

Definition 4.2. The set of atomic formulae, also called *molecules* in $L(V)$ is defined as follows:

- φ **memberOf** Γ is an atomic formula in $L(V)$
- γ **subConceptOf** Γ is an atomic formula in $L(V)$
- $\gamma[s \text{ ofType } \Delta]$ is an atomic formula in $L(V)$
- $\gamma[s \text{ impliesType } \Delta]$ is an atomic formula in $L(V)$

- $\gamma[p \text{ **impliesType** } \Gamma]$ is an atomic formula in $L(V)$
- $\phi[p \text{ **hasValue** } \psi]$ is an atomic formula in $L(V)$
- $\phi[s \text{ **hasValue** } Val]$ is an atomic formula in $L(V)$
- If α and $\beta \in V_1$ then $\alpha = \beta$ is an atomic formula in $L(V)$

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

Similar to most definitions of Description Logics, we distinguish between descriptions and formulae.

Definition 4.3. The set of descriptions in $L(V)$ is defined as follows:

- Any a-molecule or b-molecule is a description in $L(V)$.
- **true**, **false** are descriptions in $L(V)$.
- If F_1 and F_2 are descriptions in $L(V)$ and G is a b-molecule, then:
 - F_1 **and** F_2 is a description in $L(V)$.
 - F_1 **or** F_2 is a description in $L(V)$.
 - **neg** F_1 is a description in $L(V)$.
 - **forall** $Var_i G$ **implies** F_1 is a description in $L(V)$.
 - **exists** $Var_i G$ **and** F_1 is a description in $L(V)$.
- **exists** $Var_1, \dots, Var_n G_1$ **and** ... **and** G_n **and** F_1 **and** **neg** $(Var_1 = Var_2)$ **and** ... **and** **neg** $(Var_1 = Var_n)$ **and** ... **and** **neg** $(Var_{n-1} = Var_n)$ (minimal cardinality) is a description in $L(V)$.
- **forall** $Var_1, \dots, Var_{n+1} G_1$ **and** ... **and** G_{n+1} **and** F_1 **implies** $Var_1 = Var_2$ **or** ... **or** $Var_1 = Var_{n+1}$ **or** ... **or** $Var_n = Var_{n+1}$ (maximal cardinality) is a description in $L(V)$.

The *variable graph* of a description F is defined as the undirected graph having all WSML variables in F as nodes and an edge between Var_1 and Var_2 for every molecule $Var_1[p \text{ **hasValue** } Var_2]$.

In the following, F_1, F_2 stand for WSML-DL descriptions with connected, acyclic variable graphs; furthermore, the variable graphs of F_1 and F_2 can be seen as trees which share the same root node. G stands for a WSML-DL formula.

Definition 4.4. The set of WSML-DL formulae in $L(V)$ is defined as follows:

- Any atomic formula is a formula in $L(V)$.
- If α and $\beta \in V_1$ then **neg** $\alpha = \beta$ is a formula in $L(V)$.
- Any description F_1, F_2 is a formula in $L(V)$.
- F_1 **implies** F_2 is a formula in $L(V)$.
- F_1 **impliedBy** F_2 is a formula in $L(V)$.
- F_1 **equivalent** F_2 is a formula in $L(V)$.
- **forall** $Var_i(G)$ is a formula in $L(V)$.
- $Var_1[p \text{ **hasValue** } Var_2]$ **impliedBy** $Var_1[p \text{ **hasValue** } Var_3]$ **and** $Var_3[p$

`hasValue Var2]` (globally transitive attribute/relation) is a formula in $L(V)$.

- `Var1[p hasValue Var2] impliedBy Var2[p hasValue Var1]` (globally symmetric attribute/relation) is a formula in $L(V)$.
- `Var1[p hasValue Var2] impliedBy Var1[q hasValue Var2]` (globally sub-attribute/relation) is a formula in $L(V)$.
- `Var1[p hasValue Var2] impliedBy Var2[q hasValue Var1]` (globally inverse attribute/relation) is a formula in $L(V)$.

Note that the top level implication, inverse implication or equivalence is not needed. If it is omitted, a formula F is implicitly implied by **true**:

true implies F.

All variables that are not explicitly quantified are implicitly universally quantified. This entails the possibility to embed WSML-DL formulae in an outer universal quantification.

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

The following are examples of WSML-DL logical expressions:

The concept Human is defined as the disjunction between Man and Woman.

```
?x memberOf Human
  equivalent
  ?x memberOf Woman or ?x memberOf Man .
```

The concepts Man and Woman are disjoint.

```
?x memberOf Man
  implies
  neg ?x memberOf Woman .
```

Every Human has a father, which is a Human and every father is a human.

```
?x memberOf Human
  implies
  exists ?y (
    ?x[father hasValue ?y]
    and ?y memberOf Human )
  and
  forall ?y (
    ?x[father hasValue ?y]
    implies ?y memberOf Human ) .
```

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 with respect to the WSML-Flight semantics also hold with respect to 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.

Section 5.1 defines the restrictions on logical expressions in WSML-Flight. Section 5.2 outlines the differences between WSML-Core and WSML-Flight.

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

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

- We define the set of admissible head formulae $Head(V)$ as follows:
 - Any atomic formula α which does not contain the inequality symbol (\neq) or the unification operator ($=$) is in $Head(V)$.
 - Let $\alpha, \beta \in Head(V)$, then α **and** β is in $Head(V)$.
 - Given two formulae α, β such that α, β do not contain **{ implies, impliedBy, equivalent }**, the following formulae are in $Head(V)$:
 - α **implies** β , if $\beta \in Head(V)$ and $\alpha \in Head(V)$ or $\alpha \in Body(V)$
 - α **impliedBy** β , if $\alpha \in Head(V)$ and $\beta \in Head(V)$ or $\beta \in Body(V)$
 - α **equivalent** β if $\alpha \in Head(V)$ or $\alpha \in Body(V)$ and $\beta \in Head(V)$ or $\beta \in Body(V)$
- Any variable-free 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 any atomic formula α , **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 $Body(V)$.

- Given a head-formula $\beta \in \text{Head}(V)$ and a body-formula $\alpha \in \text{Body}(V)$, $\beta \text{ :- } \alpha$ 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, and (3) the safety condition holds.
- Any formula of the form $\text{!- } \alpha$ with $\alpha \in \text{Body}(V)$ is an admissible formula and is called a *constraint*.
- The Logic Programming implication symbol !- is not absolutely needed. If it is missing, a formula is in Head.

As with the general WSML logical expression syntax, <- , -> and <-> can be seen as synonyms of the keywords **implies**, **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 $A_1 \text{ and } \dots \text{ and } A_n \text{ :- } B$ are split into n different rules:
 - $A_1 \text{ :- } B$
 - ...
 - $A_n \text{ :- } B$
- Rules of the form $A_1 \text{ equivalent } A_2 \text{ :- } B$ are split into 2 rules:
 - $A_1 \text{ implies } A_2 \text{ :- } B$
 - $A_1 \text{ impliedBy } A_2 \text{ :- } B$
- Rules of the form $A_1 \text{ impliedBy } A_2 \text{ :- } B$ are transformed to:
 - $A_1 \text{ :- } A_2 \text{ and } B$
- Rules of the form $A_1 \text{ implies } A_2 \text{ :- } B$ are transformed to:
 - $A_2 \text{ :- } A_1 \text{ and } B$
- Rules of the form $A \text{ :- } B_1 \text{ and } (F \text{ or } G) \text{ and } B_2$ are split into two different rules:
 - $A \text{ :- } B_1 \text{ and } F \text{ and } B_2$
 - $A \text{ :- } B_1 \text{ and } G \text{ and } B_2$
- Rules of the form $A \text{ :- } B_1 \text{ and naf } (F \text{ and } G) \text{ and } B_2$ are split into two different rules:
 - $A \text{ :- } B_1 \text{ and naf } F \text{ and } B_2$
 - $A \text{ :- } B_1 \text{ and naf } G \text{ and } B_2$
- Rules of the form $A \text{ :- } B_1 \text{ and naf } (F \text{ or } G) \text{ and } B_2$ are transformed to:
 - $A \text{ :- } B_1 \text{ and naf } F \text{ and naf } G \text{ and } B_2$
- Rules of the form $A \text{ :- } B_1 \text{ and naf naf } F \text{ and } B_2$ are transformed to:
 - $A \text{ :- } B_1 \text{ and } F \text{ and } B_2$

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:

$p(?x) :- q(?y).$
 $a[b \text{ hasValue } ?x] :- ?x > 25.$
 $?x[\text{gender hasValue male}] :- \mathbf{naf} ?x[\text{gender hasValue female}].$

We require each WSML-Flight knowledge base to be *locally stratified*. For more details on local stratification please refer to [\[de Bruijn, 2007\]](#).

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

No human can be both male and female:

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

The brother of a parent is an uncle:

$?x[\text{uncle hasValue } ?z] \mathbf{impliedBy} ?x[\text{parent hasValue } ?y] \mathbf{and} ?y[\text{brother hasValue } ?z].$

Do not trust strangers:

$?x[\text{distrust hasValue } ?y] :- \mathbf{naf} ?x[\text{knows hasValue } ?y] \mathbf{and} ?x \mathbf{memberOf} \text{ Human and } ?y \mathbf{memberOf} \text{ Human}.$

5.2. 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 2.2](#)) 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, \dots, t_n are terms, then $r(t_1, \dots, t_n)$ is an atomic formula in $L(V)$.
- If $\alpha, \beta \in Term_{Rule}(V)$ then $\alpha = \beta$, and $\alpha \neq \beta$ are atomic formulae in $L(V)$.
- If $\alpha, \beta \in Term_{Rule}(V)$ and $\gamma \in Term(V)$ or γ is of the form $\{ \gamma_1, \dots, \gamma_n \}$ with $\gamma_1, \dots, \gamma_n \in Term_{Rule}(V)$, then:
 - α **subConceptOf** γ is an atomic formula in $L(V)$
 - α **memberOf** γ is an atomic formula in $L(V)$
 - $\alpha[\beta$ **ofType** $\gamma]$ is an atomic formula in $L(V)$
 - $\alpha[\beta$ **impliesType** $\gamma]$ is an atomic formula in $L(V)$

- $\alpha[\beta \text{ hasValue } \gamma]$ 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:

- We define the set of admissible head formulae $Head(V)$ as follows:
 - Any atomic formula α which does not contain the inequality symbol (\neq) or the unification operator ($=$) is in $Head(V)$.
 - Let $\alpha, \beta \in Head(V)$, then $\alpha \text{ and } \beta$ is in $Head(V)$.
 - Given two formulae α, β such that α, β do not contain $\{ \text{implies, impliedBy, equivalent} \}$, the following formulae are in $Head(V)$:
 - $\alpha \text{ implies } \beta$, if $\beta \in Head(V)$ and $\alpha \in Head(V)$ or $\alpha \in Body(V)$
 - $\alpha \text{ impliedBy } \beta$, if $\alpha \in Head(V)$ and $\beta \in Head(V)$ or $\beta \in Body(V)$
 - $\alpha \text{ equivalent } \beta$ if $\alpha \in Head(V)$ or $\alpha \in Body(V)$ and $\beta \in Head(V)$ or $\beta \in Body(V)$
- 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 $\alpha \in Body(V)$, $\text{naf } \alpha$ is in $Body(V)$.
 - For $\alpha, \beta \in Body(V)$, $\alpha \text{ and } \beta$ is in $Body(V)$.
 - For $\alpha, \beta \in Body(V)$, $\alpha \text{ or } \beta$ is in $Body(V)$.
 - For $\alpha, \beta \in Body(V)$, $\alpha \text{ implies } \beta$ is in $Body(V)$.
 - For $\alpha, \beta \in Body(V)$, $\alpha \text{ impliedBy } \beta$ is in $Body(V)$.
 - For $\alpha, \beta \in Body(V)$, $\alpha \text{ equivalent } \beta$ is in $Body(V)$.
 - For variables $?x_1, \dots, ?x_n$ and $\alpha \in Body(V)$, $\text{forall } ?x_1, \dots, ?x_n (\alpha)$ is in $Body(V)$.
 - For variables $?x_1, \dots, ?x_n$ and $\alpha \in Body(V)$, $\text{exists } ?x_1, \dots, ?x_n (\alpha)$ is in $Body(V)$.
- Given a head-formula $\beta \in Head(V)$ and a body-formula $\alpha \in Body(V)$, $\beta \text{ :- } \alpha$ 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.
- Any formula of the form $\text{!- } \alpha$ with $\alpha \in Body(V)$ is an admissible formula and is called a *constraint*.
- The Logic Programming implication symbol !- is not absolutely needed. If it is missing, a formula is in $Head$.

As with the general WSML logical expression syntax, \leftarrow , \rightarrow and \leftrightarrow can be seen as synonyms of the keywords **implies**, **impliedBy** and **equivalent**, respectively.

The following are examples of WSML-Rule logical expressions:

Both the father and the mother are parents:

$?x[\text{parent hasValue } ?y] \text{ :- } ?x[\text{father hasValue } ?y] \text{ or } ?x[\text{mother hasValue } ?y].$

Every person has a father:

```
?x[father hasValue f(?x)] :- ?x memberOf Person.
```

There may only be one distance between two locations, and the distance between locations *A* and *B* is the same as the distance between *B* and *A*:

```
!- distance(?location1, ?location2, ?distance1) and  
  distance(?location1, ?location2, ?distance2) and ?distance1 != distance2.
```

```
distance(?B, ?A, ?distance) :-  
  distance(?A, ?B, ?distance).
```

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 unifies the Description Logic and Logic Programming variants of WSML, namely, WSML-DL and WSML-Rule, in a principled way, under a common syntactic umbrella, inspired by First-Order Logic .

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.

Even though there is a proposal for a semantics for WSML-Full [[de Bruijn & Heymans, 2007](#)], there is no consensus in the research community as to what is the right semantics for combinations of the Description Logic (WSML-DL) ontologies and rules with nonmonotonic negation (WSML-Rule). Therefore, WSML does not specify a semantics for its Full variant.

Both the conceptual and logical expression syntax for WSML-Full completely corresponds with the WSML syntax introduced in Chapter 2.

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.

Appendix A. Human-Readable Syntax

This appendix presents the complete grammar for the WSML language. The language used to 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 variant 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

<u>all</u>	= [0x0 .. 0xffff]
<u>escape_char</u>	= '\'
<u>basechar</u>	= [0x0041 .. 0x005A] [0x0061 .. 0x007A]
<u>ideographic</u>	= [0x4E00 .. 0x9FA5] 0x3007 [0x3021 .. 0x3029]
<u>letter</u>	= <u>basechar</u> <u>ideographic</u>
<u>digit</u>	= [0x0030 .. 0x0039]
<u>combiningchar</u>	= [0x0300 .. 0x0345] [0x0360 .. 0x0361] [0x0483 .. 0x0486]
<u>extender</u>	= 0x00B7 0x02D0 0x02D1 0x0387 0x0640 0x0E46 0x0EC6 0x3005 [0x3031 .. 0x3035] [0x309D .. 0x309E] [0x30FC .. 0x30FE]
<u>alphanum</u>	= <u>digit</u> <u>letter</u>
<u>hexdigit</u>	= ['0' .. '9'] ['A' .. 'F']
<u>not_escaped_namechar</u>	= <u>letter</u> <u>digit</u> '_' <u>combiningchar</u> <u>extender</u>
<u>escaped_namechar</u>	= '\'' '\"' <u>not_escaped_namechar</u>
<u>namechar</u>	= (<u>escape_char</u> <u>escaped_namechar</u>) <u>not_escaped_namechar</u>

<u>reserved</u>	= <code>' '?' '#' '[' ']' ':' '@' '&' '=' '+' '\$' ';</code>
<u>mark</u>	= <code>' '_' '.' '!' '~' '*' '"' '(' ')'</code>
<u>escaped</u>	= <code>'%' hexdigit hexdigit</code>
<u>unreserved</u>	= <code>letter digit mark</code>
<u>scheme</u>	= <code>letter (letter digit '+' '-' ':')*</code>
<u>port</u>	= <code>digit*</code>
<u>idomainlabel</u>	= <code>alphanum ((alphanum '-') * alphanum)?</code>
<u>dec_octet</u>	= <code>digit ([0x31 .. 0x39] digit) ('1' digit digit) ('2' [0x30 .. 0x34] digit) ('25' [0x30 .. 0x35])</code>
<u>ipv4address</u>	= <code>dec_octet ':' dec_octet ':' dec_octet ':' dec_octet</code>
<u>h4</u>	= <code>hexdigit hexdigit? hexdigit? hexdigit?</code>
<u>ls32</u>	= <code>(h4 ':' h4) ipv4address</code>
<u>ipv6address</u>	= <code>((h4 ':') * h4)? ':' (h4 ':') * ls32 ((h4 ':') * h4)? ':' h4 ((h4 ':') * h4)? ':'</code>
<u>ipv6reference</u>	= <code>'[' ipv6address ']'</code>
<u>ucschar</u>	= <code>[0xA0 .. 0xD7FF] [0xF900 .. 0xFDCF] [0xFDF0 .. 0xFFEF]</code>
<u>iunreserved</u>	= <code>unreserved ucschar</code>
<u>ipchar</u>	= <code>iunreserved escaped ':' '@' '&' '=' '+' '\$' ';</code>
<u>isegment</u>	= <code>ipchar*</code>
<u>ipath_segments</u>	= <code>isegment ('/' isegment)*</code>
<u>iuserinfo</u>	= <code>(iunreserved escaped ':' '@' '&' '=' '+' '\$' ';')*</code>
<u>iqualified</u>	= <code>(':' idomainlabel) * ':' ?</code>
<u>ihostname</u>	= <code>idomainlabel iqualified</code>
<u>ihost</u>	= <code>(ipv6reference ipv4address ihostname)?</code>
<u>iauthority</u>	= <code>(iuserinfo '@')? ihost (':' port)?</code>
<u>iabs_path</u>	= <code>'/' ipath_segments</code>
<u>inet_path</u>	= <code>'// ' iauthority (iabs_path)?</code>
<u>irel_path</u>	= <code>ipath_segments</code>
<u>ihier_part</u>	= <code>inet_path iabs_path irel_path</code>
<u>iprivate</u>	= <code>[0xE000 .. 0xF8FF]</code>
<u>iquery</u>	= <code>(ipchar iprivate '?' '?')*</code>
<u>ifragment</u>	= <code>(ipchar '?' '?')*</code>
<u>iri_f</u>	= <code>scheme ':' ihier_part ('?' iquery)? ('#' ifragment)?</code>
<u>tab</u>	= <code>9</code>
<u>cr</u>	= <code>13</code>
<u>lf</u>	= <code>10</code>
<u>eol</u>	= <code>cr lf cr lf</code>
<u>squote</u>	= <code>'</code>
<u>dquote</u>	= <code>''</code>
<u>not_cr_lf</u>	= <code>[all - [cr+ lf]]</code>
<u>escaped_char</u>	= <code>escape_char all</code>
<u>not_escape_char_not_dquote</u>	= <code>[all - ['' + escape_char]]</code>
<u>string_content</u>	= <code>escaped_char not_escape_char_not_dquote</code>
<u>long_comment_content</u>	= <code>[all - '/'] [all - '*'] '/'</code>
<u>long_comment</u>	= <code>'/*' long_comment_content* '*/'</code>

<u>begin_comment</u>	= <code>/// </code>
<u>short_comment</u>	= <u>begin_comment</u> <code>not_cr lf* eol</code>
<u>comment</u>	= <u>short_comment</u> <u>long_comment</u>
<u>blank</u>	= (<code>' ' tab eol</code>) ⁺
<u>qmark</u>	= <code>'?'</code>
<u>luridel</u>	= <code>'_''</code>
<u>ruridel</u>	= <code>''''</code>

States

- default
- in_ws_or_goal

Tokens

<u>t_comment</u>	= <u>comment</u>
<u>t_blank</u>	= <u>blank</u>
<u>at</u>	= <code>@'</code>
<u>arrow</u>	= <code>'=>'</code>
<u>comma</u>	= <code>','</code>
<u>t_pipe</u>	= <code>' '</code>
<u>endpoint</u>	= <code>':'</code> <u>blank</u>
<u>lpar</u>	= <code>'('</code>
<u>rpar</u>	= <code>)'</code>
<u>lbracket</u>	= <code>'['</code>
<u>rbracket</u>	= <code>']'</code>
<u>lbrace</u>	= <code>'{'</code>
<u>rbrace</u>	= <code>'}'</code>
<u>hash</u>	= <code>'#'</code>
<u>t_and</u>	= <code>'and'</code>
<u>t_or</u>	= <code>'or'</code>
<u>t_implies</u>	= <code>'implies' '->'</code>
<u>t_implied by</u>	= <code>'impliedBy' '<-'</code>
<u>t_equivalent</u>	= <code>'equivalent' '<->'</code>
<u>t_implied by lp</u>	= <code>':-'</code>
<u>t_constraint</u>	= <code>'!'</code>
<u>t_not</u>	= <code>'neg' 'naf'</code>
<u>t_exists</u>	= <code>'exists'</code>
<u>t_forall</u>	= <code>'forall'</code>
<u>t_univfalse</u>	= <code>'false'</code>
<u>t_univtrue</u>	= <code>'true'</code>
<u>gt</u>	= <code>'>'</code>
<u>lt</u>	= <code>'<'</code>
<u>gte</u>	= <code>'>='</code>
<u>lte</u>	= <code>'<='</code>
<u>equal</u>	= <code>'='</code>
<u>unequal</u>	= <code>'!='</code>
<u>add_op</u>	= <code>'+'</code>

	<u>sub_op</u>	= '.'
	<u>star</u>	= '*'
	<u>div_op</u>	= '/'
	<u>t_annotations</u>	= 'annotations'
	<u>t_endannotations</u>	= 'endAnnotations'
	<u>t_assumption</u>	= 'assumption'
	<u>t_axiom</u>	= 'axiom'
{ in_ws_or_goal }		
	<u>t_innercapability</u>	= 'capability'
	<u>t_capability</u>	= 'capability'
	<u>t_choreography</u>	= 'choreography'
	<u>t_statesignature</u>	= 'stateSignature'
	<u>t_static</u>	= 'static'
	<u>t_in</u>	= 'in'
	<u>t_out</u>	= 'out'
	<u>t_shared</u>	= 'shared'
	<u>t_controlled</u>	= 'controlled'
	<u>t_withgrounding</u>	= 'withGrounding'
	<u>t_transitionrules</u>	= 'transitionRules'
	<u>t_if</u>	= 'if'
	<u>t_then</u>	= 'then'
	<u>t_endif</u>	= 'endif'
	<u>t_with</u>	= 'with'
	<u>t_endforall</u>	= 'endForall'
	<u>t_choose</u>	= 'choose'
	<u>t_endchoose</u>	= 'endChoose'
	<u>t_do</u>	= 'do'
	<u>t_add</u>	= 'add'
	<u>t_delete</u>	= 'delete'
	<u>t_update</u>	= 'update'
	<u>t_concept</u>	= 'concept'
	<u>t_definedby</u>	= 'definedBy'
	<u>t_effect</u>	= 'effect'
{ in_ws_or_goal -> default , default }		
	<u>t_ggmediator</u>	= 'ggMediator'
{ default -> in_ws_or_goal , in_ws_or_goal }		
	<u>t_goal</u>	= 'goal'
	<u>t_hasvalue</u>	= 'hasValue'
	<u>t_impliestype</u>	= 'impliesType'
	<u>t_importontology</u>	= 'importsOntology'
	<u>t_instance</u>	= 'instance'
{ in_ws_or_goal }		
	<u>t_innerinterface</u>	= 'interface'
	<u>t_interface</u>	= 'interface'
	<u>t_inverseof</u>	= 'inverseOf'
	<u>t_subattributeof</u>	= 'subAttributeOf'
	<u>t_memberof</u>	= 'memberOf'

	<u>t_namespace</u>	= 'namespace'
	<u>t_nfp</u>	= 'nonFunctionalProperty' 'nfp'
	<u>t_ofType</u>	= 'ofType'
{ in_ws_or_goal -> default , default }		
{ in_ws_or_goal -> default , default }	<u>t_ontology</u>	= 'ontology'
	<u>t_oomediator</u>	= 'ooMediator'
	<u>t_orchestration</u>	= 'orchestration'
	<u>t_postcondition</u>	= 'postcondition'
	<u>t_precondition</u>	= 'precondition'
	<u>t_reflexive</u>	= 'reflexive'
	<u>t_relation</u>	= 'relation'
	<u>t_relation_instance</u>	= 'relationInstance'
	<u>t_sharedvariable</u>	= 'sharedVariables'
	<u>t_source</u>	= 'source'
	<u>t_subconcept</u>	= 'subConceptOf'
	<u>t_subrelation</u>	= 'subRelationOf'
	<u>t_symmetric</u>	= 'symmetric'
	<u>t_target</u>	= 'target'
	<u>t_transitive</u>	= 'transitive'
	<u>t_usemediator</u>	= 'usesMediator'
	<u>t_useservice</u>	= 'usesService'
{ default -> in_ws_or_goal , in_ws_or_goal }		
{ in_ws_or_goal -> default , default }	<u>t_webservice</u>	= 'webService'
	<u>t_wgmediator</u>	= 'wgMediator'
	<u>t_wsmlvariant</u>	= 'wsmlVariant'
{ in_ws_or_goal -> default , default }		
	<u>t_wwmediator</u>	= 'wwMediator'
	<u>variable</u>	= <u>qmark</u> <u>alphanum</u> ⁺
	<u>anonymous</u>	= <u>'_#'</u>
	<u>nb_anonymous</u>	= <u>'_#'</u> <u>digit</u> ⁺
	<u>pos_integer</u>	= <u>digit</u> ⁺
	<u>pos_decimal</u>	= <u>digit</u> ⁺ <u>'.'</u> <u>digit</u> ⁺
	<u>string</u>	= <u>dquote</u> <u>string content</u> [*] <u>dquote</u>
	<u>full iri</u>	= <u>luridel</u> <u>iri f</u> <u>ruridel</u>
	<u>name</u>	= (<u>letter</u> <u>'_'</u>) <u>namechar</u> [*]

Ignored Tokens

- t_blank
- t_comment

Productions

wsml = wsmlvariant? namespace? definition*

<u>wsmmlvariant</u>	=	<u>t_wsmmlvariant</u> <u>full_iri</u>
<u>namespace</u>	=	<u>t_namespace</u> <u>prefixdefinitionlist</u>
<u>prefixdefinitionlist</u>	=	{ <u>defaultns</u> } <u>full_iri</u> { <u>prefixdefinitionlist</u> } <u>lbrace</u> <u>prefixdefinition</u> <u>moreprefixdefinitions*</u> <u>rbrace</u>
<u>prefixdefinition</u>	=	{ <u>namespacedef</u> } <u>name</u> <u>full_iri</u> { <u>default</u> } <u>full_iri</u>
<u>moreprefixdefinitions</u>	=	<u>comma</u> <u>prefixdefinition</u>
<u>definition</u>	=	{ <u>goal</u> } <u>goal</u> { <u>ontology</u> } <u>ontology</u> { <u>webservice</u> } <u>webservice</u> { <u>mediator</u> } <u>mediator</u> { <u>capability</u> } <u>capability</u> { <u>interface</u> } <u>interface</u>
<u>header</u>	=	{ <u>annotations</u> } <u>annotations</u> { <u>importsontology</u> } <u>importsontology</u> { <u>usesmediator</u> } <u>usesmediator</u>
<u>usesmediator</u>	=	<u>t_usemediator</u> <u>irilist</u>
<u>importsontology</u>	=	<u>t_importontology</u> <u>irilist</u>
<u>annotations</u>	=	<u>t_annotations</u> <u>annotationvalue*</u> <u>t_endannotations</u>
<u>mediator</u>	=	{ <u>oomediator</u> } <u>oomediator</u> { <u>ggmediator</u> } <u>ggmediator</u> { <u>wgmediator</u> } <u>wgmediator</u> { <u>wwmediator</u> } <u>wwmediator</u>
<u>baseheader</u>	=	{ <u>base</u> } <u>annotations</u> { <u>imports</u> } <u>importsontology</u>
<u>oomediator</u>	=	<u>t_oomediator</u> <u>iri?</u> <u>baseheader*</u> <u>nfp*</u> <u>sources?</u> <u>target?</u> <u>use_service?</u>
<u>ggmediator</u>	=	<u>t_ggmediator</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>source?</u> <u>target?</u> <u>use_service?</u>
<u>wgmediator</u>	=	<u>t_wgmediator</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>sources?</u> <u>target?</u> <u>use_service?</u>
<u>wwmediator</u>	=	<u>t_wwmediator</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>source?</u> <u>target?</u> <u>use_service?</u>
<u>use_service</u>	=	<u>t_useservice</u> <u>iri</u>
<u>source</u>	=	<u>t_source</u> <u>iri</u>
<u>msources</u>	=	<u>t_source</u> <u>lbrace</u> <u>iri?</u> <u>moreiris*</u> <u>rbrace</u>
<u>sources</u>	=	{ <u>single</u> } <u>source</u> { <u>multiple</u> } <u>msources</u>
<u>target</u>	=	<u>t_target</u> <u>iri</u>
<u>goal</u>	=	<u>t_goal</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>innercapability?</u> <u>innerinterfaces*</u>
<u>webservice</u>	=	<u>t_webservice</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>innercapability?</u> <u>innerinterfaces*</u>
<u>innercapability</u>	=	<u>t_innercapability</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>sharedvardef?</u> <u>pre_post ass or eff*</u>
<u>capability</u>	=	<u>t_capability</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>sharedvardef?</u> <u>pre_post ass or eff*</u>

<u>sharedvardef</u>	=	<u>t_sharedvariable</u> <u>variablelist</u>
<u>pre_post_ass_or_eff</u>	=	{precondition} <u>t_precondition</u> <u>axiomdefinition</u> {postcondition} <u>t_postcondition</u> <u>axiomdefinition</u> {assumption} <u>t_assumption</u> <u>axiomdefinition</u> {effect} <u>t_effect</u> <u>axiomdefinition</u>
<u>innerinterfaces</u>	=	{single} <u>innerinterface</u> {multiple} <u>minnerinterfaces</u>
<u>minnerinterfaces</u>	=	<u>t_innerinterface</u> <u>lbrace</u> <u>iri</u> <u>moreiris*</u> <u>rbrace</u>
<u>interface</u>	=	<u>t_interface</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>choreography?</u> <u>orchestration?</u>
<u>innerinterface</u>	=	<u>t_innerinterface</u> <u>iri?</u> <u>header*</u> <u>nfp*</u> <u>choreography?</u> <u>orchestration?</u>
<u>choreography</u>	=	<u>t_choreography</u> <u>iri?</u> <u>header*</u> <u>statesignature?</u> <u>transitions?</u>
<u>statesignature</u>	=	<u>t_statesignature</u> <u>id?</u> <u>header*</u> <u>mode*</u>
<u>mode</u>	=	{grounded_mode} <u>grounded_mode</u> {ungrounded_mode} <u>ungrounded_mode</u>
<u>grounded_mode</u>	=	{in} <u>t_in</u> <u>grounded mode entry list</u> {out} <u>t_out</u> <u>grounded mode entry list</u> {shared} <u>t_shared</u> <u>grounded mode entry list</u>
<u>ungrounded_mode</u>	=	{static} <u>t_static</u> <u>ungrounded mode entry list</u> {controlled} <u>t_controlled</u> <u>ungrounded mode entry list</u>
<u>grounded_mode_entry_list</u>	=	{mode_entry} <u>grounded mode entry</u> {mode_entry_list} <u>grounded mode entry comma</u> <u>grounded mode entry list</u>
<u>ungrounded_mode_entry_list</u>	=	{mode_entry} <u>ungrounded mode entry</u> {mode_entry_list} <u>ungrounded mode entry comma</u> <u>ungrounded mode entry list</u>
<u>grounded_mode_entry</u>	=	{default} <u>iri</u> <u>grounding</u> {concept} <u>t_concept</u> <u>iri</u> <u>grounding</u> {relation} <u>t_relation</u> <u>iri</u> <u>grounding</u>
<u>ungrounded_mode_entry</u>	=	{default} <u>iri</u> {concept} <u>t_concept</u> <u>iri</u> {relation} <u>t_relation</u> <u>iri</u>
<u>grounding</u>	=	<u>t_withgrounding</u> <u>irilist</u>
<u>transitions</u>	=	<u>t_transitionrules</u> <u>id?</u> <u>annotations?</u> <u>rule*</u>
<u>rule</u>	=	{if} <u>t_if</u> <u>condition</u> <u>t_then</u> <u>rule+</u> <u>t_endif</u> <u>pipedef rule?</u> {forall} <u>t_forall</u> <u>variables</u> <u>t_with</u> <u>condition</u> <u>t_do</u> <u>rule+</u> <u>t_endforall</u> <u>pipedef rule?</u> {choose} <u>t_choose</u> <u>variables</u> <u>t_with</u> <u>condition</u> <u>t_do</u> <u>rule+</u> <u>t_endchoose</u> <u>pipedef rule?</u> {update} <u>updaterule</u>
<u>condition</u>	=	{restricted_le} <u>expr</u>
<u>pipedef rule</u>	=	{pipedef_rule} <u>t_pipe</u> <u>rule</u>
<u>updaterule</u>	=	<u>modifier</u> <u>lpar</u> <u>fact</u> <u>rpar</u>

<u>modifier</u>	=	{add} t <u>add</u> {delete} t <u>delete</u> {update} t <u>update</u>
<u>fact</u>	=	{fact_preferred} term attr fact? t <u>memberof</u> <u>termlist</u> <u>fact_update</u> ? {fact_nonpreferred} term t <u>memberof</u> <u>termlist</u> <u>fact_update</u> ? attr fact {fact_molecule} term attr fact {fact_relation} at id lpar <u>term_updates</u> rpar
<u>fact_update</u>	=	arrow <u>termlist</u>
<u>attr_fact</u>	=	lbracket attr fact list rbracket
<u>attr_fact_list</u>	=	{attr_relation} term t <u>hasvalue</u> <u>termlist</u> <u>fact_update</u> ? attr fact list comma term t <u>hasvalue</u> <u>termlist</u> <u>fact_update</u> ?
<u>term_updates</u>	=	{one_param} term <u>update</u> {more_params} term <u>update</u> comma <u>term_updates</u>
<u>term_update</u>	=	{single} term {move} [oldterm]: term arrow [newterm]: term
<u>new_term</u>	=	{new_term} arrow term
<u>orchestration</u>	=	t <u>orchestration</u> iri?
<u>nfp</u>	=	t nfp <u>attributevaluenfp</u> <u>log_definition</u> nfp?
<u>attributevaluenfp</u>	=	id t <u>hasvalue</u> <u>valuelistnfp</u> <u>annotations</u> ?
<u>valuenfp</u>	=	{normal} <u>basevalue</u> {var} <u>variable</u>
<u>valuelistnfp</u>	=	{single} <u>valuenfp</u> {multi} lbrace <u>valuenfp</u> <u>morenfpvalues</u> * rbrace
<u>morenfpvalues</u>	=	comma <u>valuenfp</u>
<u>ontology</u>	=	t <u>ontology</u> iri? <u>header</u> * <u>ontology_element</u> *
<u>ontology_element</u>	=	{concept} <u>concept</u> {instance} <u>instance</u> {relation} <u>relation</u> {relationinstance} <u>relationinstance</u> {axiom} <u>axiom</u>
<u>concept</u>	=	t <u>concept</u> id <u>superconcept</u> ? <u>annotations</u> ? <u>attribute</u> *
<u>superconcept</u>	=	t <u>subconcept</u> idlist
<u>att_type</u>	=	{open_world} t <u>oftype</u> {closed_world} t <u>impliestype</u>
<u>attribute</u>	=	id <u>attributefeature</u> * att type <u>cardinality</u> ? idlist <u>annotations</u> ?
<u>cardinality</u>	=	lpar pos integer <u>cardinality_number</u> ? rpar
<u>cardinality_number</u>	=	{finite_cardinality} pos integer {infinite_cardinality} star
<u>attributefeature</u>	=	{transitive} t <u>transitive</u> {symmetric} t <u>symmetric</u> {inverse} t <u>inverseof</u> lpar id rpar

	{sub_attribute_of} <u>t_subattributeof</u> <u>lpar</u> <u>id</u> <u>rpar</u>
	{reflexive} <u>t_reflexive</u>
<u>instance</u>	= <u>t_instance</u> <u>id?</u> <u>memberof?</u> <u>annotations?</u> <u>attributevalue*</u>
<u>memberof</u>	= <u>t_memberof</u> <u>idlist</u>
<u>annotationvalue</u>	= <u>id</u> <u>t_hasvalue</u> <u>basevaluelist</u>
<u>attributevalue</u>	= <u>id</u> <u>t_hasvalue</u> <u>valuelist</u>
<u>relation</u>	= <u>t_relation</u> <u>id?</u> <u>paramdef</u> <u>superrelation?</u> <u>annotations?</u>
<u>paramdef</u>	= {aritydef} <u>arity</u> {typingdef} <u>paramtyping</u>
<u>paramtyping</u>	= <u>lpar</u> <u>paramtype</u> <u>moreparamtype*</u> <u>rpar</u>
<u>paramtype</u>	= <u>att</u> <u>type</u> <u>idlist</u>
<u>moreparamtype</u>	= <u>comma</u> <u>paramtype</u>
<u>superrelation</u>	= <u>t_subrelation</u> <u>idlist</u>
<u>arity</u>	= <u>div_op</u> <u>pos</u> <u>integer</u>
<u>relationinstance</u>	= <u>t_relation</u> <u>instance</u> [<u>name</u>]: <u>id?</u> [<u>relation</u>]: <u>id</u> <u>lpar</u> <u>value</u> <u>morevalues*</u> <u>rpar</u> <u>annotations?</u>
<u>axiom</u>	= <u>t_axiom</u> <u>axiomdefinition</u>
<u>axiomdefinition</u>	= {annotations_axiom} <u>id?</u> <u>annotations?</u> {defined_axiom} <u>id?</u> <u>annotations?</u> <u>log_definition</u>
<u>log_definition</u>	= <u>t_definedby</u> <u>log_expr+</u>
<u>log_definition_nfp</u>	= <u>t_definedby</u> <u>log_expr+</u>
<u>log_expr</u>	= {lp_rule} [<u>head</u>]: <u>expr</u> <u>t_implied_by</u> <u>lp</u> [<u>body</u>]: <u>expr</u> <u>endpoint</u> {constraint} <u>t_constraint</u> <u>expr</u> <u>endpoint</u> {other_expression} <u>expr</u> <u>endpoint</u>
<u>expr</u>	= {implication} <u>expr</u> <u>imply_op</u> <u>disjunction</u> {disjunction} <u>disjunction</u>
<u>disjunction</u>	= {conjunction} <u>conjunction</u> <u>disjunction</u> <u>t_or</u> <u>conjunction</u>
<u>conjunction</u>	= {subexpr} <u>subexpr</u> <u>conjunction</u> <u>t_and</u> <u>subexpr</u>
<u>subexpr</u>	= {negated} <u>t_not</u> <u>subexpr</u> {simple} <u>simple</u> {complex} <u>lpar</u> <u>expr</u> <u>rpar</u> {quantified} <u>quantified</u>
<u>quantified</u>	= <u>quantifier</u> <u>key</u> <u>variables</u> <u>lpar</u> <u>expr</u> <u>rpar</u>
<u>simple</u>	= {molecule} <u>molecule</u> {comparison} <u>comparison</u> {atom} <u>term</u> {universal_truth} <u>t_univtrue</u> {universal_falsehood} <u>t_univfalse</u>
<u>molecule</u>	= {concept_molecule_preferred} <u>term</u> <u>attr_specification?</u> <u>cpt</u> <u>op</u> <u>termlist</u>

	{concept_molecule_nonpreferred} <u>term cpt_op termlist</u> <u>attr specification</u>
	{attribute_molecule} <u>term attr specification</u>
<u>attr_specification</u>	= <u>lbracket attr rel list rbracket</u>
<u>attr_rel_list</u>	= {attr_relation} <u>attr relation</u> <u>attr_rel_list comma attr relation</u>
<u>attr_relation</u>	= {attr_def} <u>term attr def op termlist</u> {attr_val} <u>term t hasvalue termlist</u>
<u>comparison</u>	= [left]: <u>term comp op [right]: term</u>
<u>functionsymbol</u>	= {parametrized} <u>id lpar terms? rpar</u> {math} <u>lpar arith val rpar</u>
<u>arith_val</u>	= <u>mult_val</u> {addition} <u>arith_val arith op mult_val</u> {semisimple1_addition} <u>term arith op mult_val</u> {semisimple2_addition} <u>arith_val arith op term</u> {simple_addition} <u>[a]: term arith op [b]: term</u>
<u>mult_val</u>	= <u>[a]: term mul_op [b]: term</u> {multiplication} <u>mult_val mul_op term</u>
<u>math_op</u>	= {arith} <u>arith op</u> {mult} <u>mul_op</u>
<u>arith_op</u>	= {add} <u>add_op</u> {sub} <u>sub_op</u>
<u>mul_op</u>	= {mul} <u>star</u> {div} <u>div_op</u>
<u>comp_op</u>	= {gt} <u>gt</u> {lt} <u>lt</u> {gte} <u>gte</u> {lte} <u>lte</u> {equal} <u>equal</u> {unequal} <u>unequal</u>
<u>cpt_op</u>	= {memberof} <u>t memberof</u> {subconceptof} <u>t subconcept</u>
<u>quantifier_key</u>	= {forall} <u>t forall</u> {exists} <u>t exists</u>
<u>attr_def_op</u>	= {oftype} <u>t oftype</u> {impliestype} <u>t impliestype</u>
<u>imply_op</u>	= {implies} <u>t implies</u> {impliedby} <u>t implied by</u> {equivalent} <u>t equivalent</u>
<u>prefix</u>	= <u>name hash</u>
<u>compacturi</u>	= {any} <u>prefix? name</u> {localkeyword} <u>prefix anykeyword</u>
<u>anykeyword</u>	= {add} <u>t add</u> {and} <u>t and</u> {annotations} <u>t annotations</u>

{assumption}	<u>t_assumption</u>
{axiom}	<u>t_axiom</u>
{capability}	<u>t_capability</u>
{choreography}	<u>t_choreography</u>
{choose}	<u>t_choose</u>
{concept}	<u>t_concept</u>
{constraint}	<u>t_constraint</u>
{controlled}	<u>t_controlled</u>
{delete}	<u>t_delete</u>
{definedby}	<u>t_definedby</u>
{do}	<u>t_do</u>
{effect}	<u>t_effect</u>
{end_annotations}	<u>t_endannotations</u>
{endchoose}	<u>t_endchoose</u>
{endif}	<u>t_endif</u>
{endforall}	<u>t_endforall</u>
{equivalent}	<u>t_equivalent</u>
{exists}	<u>t_exists</u>
{forall}	<u>t_forall</u>
{ggmediator}	<u>t_ggmediator</u>
{goal}	<u>t_goal</u>
{hasvalue}	<u>t_hasvalue</u>
{implies}	<u>t_implies</u>
{implied_by}	<u>t_implied_by</u>
{implied_by_lp}	<u>t_implied_by_lp</u>
{impliestype}	<u>t_impliestype</u>
{importontology}	<u>t_importontology</u>
{if}	<u>t_if</u>
{in}	<u>t_in</u>
{instance}	<u>t_instance</u>
{interface}	<u>t_interface</u>
{innerinterface}	<u>t_innerinterface</u>
{inverseof}	<u>t_inverseof</u>
{innercapability}	<u>t_innercapability</u>
{memberof}	<u>t_memberof</u>
{namespace}	<u>t_namespace</u>
{nfp}	<u>t_nfp</u>
{not}	<u>t_not</u>
{or}	<u>t_or</u>
{out}	<u>t_out</u>
{oftype}	<u>t_oftype</u>
{ontology}	<u>t_ontology</u>
{oomediator}	<u>t_oomediator</u>
{orchestration}	<u>t_orchestration</u>
{postcondition}	<u>t_postcondition</u>
{precondition}	<u>t_precondition</u>
{reflexive}	<u>t_reflexive</u>
{relation}	<u>t_relation</u>

	{relation_instance}	t <u>relation_instance</u>
	{shared}	t <u>shared</u>
	{sharedvariable}	t <u>sharedvariable</u>
	{source}	t <u>source</u>
	{statesignature}	t <u>statesignature</u>
	{static}	t <u>static</u>
	{subattribute}	t <u>subattributeof</u>
	{subconcept}	t <u>subconcept</u>
	{subrelation}	t <u>subrelation</u>
	{symmetric}	t <u>symmetric</u>
	{target}	t <u>target</u>
	{then}	t <u>then</u>
	{transitionrules}	t <u>transitionrules</u>
	{transitive}	t <u>transitive</u>
	{univfalse}	t <u>univfalse</u>
	{univtrue}	t <u>univtrue</u>
	{update}	t <u>update</u>
	{usemediator}	t <u>usemediator</u>
	{useservice}	t <u>useservice</u>
	{webservice}	t <u>webservice</u>
	{wgmediator}	t <u>wgmediator</u>
	{with}	t <u>with</u>
	{withgrounding}	t <u>withgrounding</u>
	{wsmlvariant}	t <u>wsmlvariant</u>
	{wwmediator}	t <u>wwmediator</u>
<u>iri</u>	= {iri}	<u>full_iri</u>
	{compacturi}	<u>compacturi</u>
<u>id</u>	= {iri}	<u>iri</u>
	{anonymous}	<u>anonymous</u>
<u>irilist</u>	= {iri}	<u>iri</u>
	{irilist}	<u>lbrace iri moreiris* rbrace</u>
<u>moreiris</u>	= <u>comma</u>	<u>iri</u>
<u>idlist</u>	= {id}	<u>id</u>
	{idlist}	<u>lbrace id moreids* rbrace</u>
	{idempty}	<u>lbrace rbrace</u>
<u>moreids</u>	= <u>comma</u>	<u>id</u>
<u>morebasevalues</u>	= <u>comma</u>	<u>basevalue</u>
<u>basevalue</u>	= {normal}	<u>datavalue</u>
	{term}	<u>id</u>
<u>basevaluelist</u>	= {single}	<u>basevalue</u>
	{multi}	<u>lbrace basevalue morebasevalues* rbrace</u>
<u>value</u>	= {identifier}	<u>id</u>
	{term}	<u>functionsymbol</u>
	{simple}	<u>numberorstring</u>
<u>numberorstring</u>	= {numeric}	<u>number</u>
	{string}	<u>string</u>

<u>datavalue</u>	=	{ numorstring }	<u>numberorstring</u>				
			{ constructeddatavalue }	<u>constructeddatavalue</u>			
<u>numberorstrings</u>	=	<u>numberorstring</u>	<u>morenumberorstrings</u> *				
<u>morenumberorstrings</u>	=	<u>comma</u>	<u>numberorstring</u>				
<u>constructeddatavalue</u>	=	<u>iri</u>	<u>lpar</u>	<u>numberorstrings</u>	<u>rpar</u>		
<u>valuelist</u>	=	{ term }	<u>value</u>				
			{ valuelist }	<u>lbrace</u>	<u>value</u>	<u>morevalues</u> *	<u>rbrace</u>
<u>morevalues</u>	=	<u>comma</u>	<u>value</u>				
<u>term</u>	=	{ data }	<u>value</u>				
			{ var }	<u>variable</u>			
			{ nb_anonymous }	<u>nb_anonymous</u>			
<u>terms</u>	=	{ term }	<u>term</u>				
			<u>terms</u>	<u>comma</u>	<u>term</u>		
<u>termlist</u>	=	{ term }	<u>term</u>				
			<u>lbrace</u>	<u>terms</u>	<u>rbrace</u>		
<u>variables</u>	=	{ variable }	<u>variable</u>				
			<u>variables</u>	<u>comma</u>	<u>variable</u>		
<u>variablelist</u>	=	{ variable }	<u>variable</u>				
			{ variable_list }	<u>lbrace</u>	<u>variables</u>	<u>rbrace</u>	
<u>integer</u>	=	<u>sub</u>	<u>op?</u>	<u>pos</u>	<u>integer</u>		
<u>decimal</u>	=	<u>sub</u>	<u>op?</u>	<u>pos</u>	<u>decimal</u>		
<u>number</u>	=	{ integer }	<u>integer</u>				
			{ decimal }	<u>decimal</u>			

A.2. Example of the Human-Readable Syntax

wsmIVariant _"http://www.wsmo.org/wsmo/wsmo-syntax/wsmo-rule"

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

/*****

*** ONTOLOGY**

*****/

ontology _"http://www.example.org/ontologies/example"

annotations

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://uibk.ac.at/~c703240/foaf.rdf",

_"http://uibk.ac.at/~c703239/foaf.rdf",

_"http://uibk.ac.at/~c703319/various/foaf.rdf" }

dc#date **hasValue** xsd#date(2004,11,22)

```

    dc#format hasValue "text/html"
    dc#language hasValue "en-US"
    dc#rights hasValue "_http://www.deri.org/privacy.html"
    wsmml#version hasValue "$Revision: 1.17 $"
endAnnotations

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
  annotations
    dc#description hasValue "concept of a human being"
  endAnnotations
  hasName ofType foaf#name
  hasRelative symmetric impliesType Human
  hasAncestor transitive impliesType Human
  hasParent inverseOf(hasChild) subAttributeOf(hasAncestor) impliesType Human
    hasMother ofType FemaleHuman
    hasMother impliesType Mother
  hasChild subAttributeOf(hasRelative) impliesType Human
  hasWeightInKG ofType (1) xsd#float
  hasBirthdate ofType (1) xsd#date
  dateOfDeath ofType (0 1) xsd#date
  hasBirthplace ofType (1) loc#location
  isMarriedTo symmetric impliesType (0 1) Human
  hasCitizenship ofType oo#country
  isAlive ofType (1) xsd#boolean
    annotations
      dc#relation hasValue {IsAlive}
    endAnnotations

relation ageOfHuman (ofType Human, ofType _integer)
  annotations
    dc#relation hasValue {FunctionalDependencyAge}
  endAnnotations

axiom IsAlive
  definedBy
    ?x[isAlive hasValue xsd#boolean("true")] :-
      naf ?x[hasObit hasValue ?obit] memberOf Human.
    ?x[isAlive hasValue xsd#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

```

```

annotations
  dc#relation hasValue ManDisjointWoman
endAnnotations

concept Woman subConceptOf Human
annotations
  dc#relation hasValue ManDisjointWoman
endAnnotations

/*
 * 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
annotations
  dc#description hasValue "Human with at least one child"
endAnnotations
  hasChild impliesType (1 *) Human

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

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

axiom ValidChild
annotations
  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."
endAnnotations
definedBy
  !- ?x memberOf Child and ageOfHuman(?x, ?age)
  and ?age > 14.

```

```

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

concept Boy
  annotations
    dc#relation hasValue {ABoy,CompletenessOfChildren}
  endAnnotations

/*
 * 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
    ?x memberOf Boy equivalent ?x memberOf Man and ?x memberOf Child.

/*
 * This axiom 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}
  annotations
    dc#description hasValue "Mary is parent of the twins Paul and Susan"
  endAnnotations
  hasName hasValue "Maria Smith"
  hasBirthdate hasValue xsd#date(1949,09,12)
  hasChild hasValue { Paul, Susan }

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

instance Susan memberOf Woman
  hasName hasValue "Susan Jones"
  hasBirthdate hasValue xsd#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 */
    hasWeightInKG hasValue 3.52
    hasBirthdate hasValue xsd#date(1977,10,21)

    relationInstance ageOfHuman(George, 1)

/*****
* WEBSERVICE
*****/
webService _"http://example.org/Germany/BirthRegistration"
  annotations
    dc#title hasValue "Birth registration service for Germany"
    dc#type hasValue _"http://www.wsmo.org/TR/d2/v1.2/#services"
    wsm#version hasValue "$Revision: 1.17 $"
  endAnnotations

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

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

  nonFunctionalProperty
    price hasValue ?price
    annotations
      dc#description hasValue "If the client is older than 60 or younger than 10 years old the
        invocation price is lower than 10 euro"
    endAnnotations
    definedBy
      ?client[age hasValue ?age] memberOf human and ?age[amount hasValue ?years, units hasValue yearsDura
        memberOf age and (greaterEqual(?years, 60) or lessEqual(?years, 10))
      implies ?price[hasAmount hasValue ?amount, hasCurrency hasValue Euro] memberOf
        absolutePrice and lessEqual(?amount, 10).

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

    assumption
      annotations
        dc#description hasValue "The child is not dead"
      endAnnotations
      definedBy
        ?child memberOf Child

```

and naf ?child[hasObit **hasValue** ?x].

effect

annotations

dc:description **hasValue** "After the registration the child
is a German citizen"

endAnnotations

definedBy

?child **memberOf** Child
and ?child[hasCitizenship **hasValue** oo#de].

interface

choreography _"http://example.org/tobedone"

orchestration _"http://example.org/tobedone"

/*****

*** GOAL**

*****/

goal _"http://example.org/Germany/GetCitizenship"

annotations

dc:title **hasValue** "Goal of getting a citizenship within Germany"
dc:type **hasValue** _"http://www.wsmo.org/TR/d2/v1.2/#goals"
wsmi#version **hasValue** "\$Revision: 1.17 \$"

endAnnotations

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

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

capability

sharedVariables ?human

effect havingACitizenship

annotations

dc:description **hasValue** "This goal expresses the general
desire of becoming a citizen of Germany."

endAnnotations

definedBy

?human **memberOf** Human[hasCitizenship **hasValue** oo#de] .

goal _"http://example.org/Germany/RegisterGeorge"

annotations

dc:title **hasValue** "Goal of getting a Registration for Paul's son George"
dc:type **hasValue** _"http://www.wsmo.org/TR/d2/v1.2/#goals"
wsmi#version **hasValue** "\$Revision: 1.17 \$"

endAnnotations

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

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

capability

effect havingRegistrationForGeorge

annotations

dc:description **hasValue** "This goal expresses Paul's desire

```

                to register his son with the German birth registration board."
            endAnnotations
            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"
    annotations
        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"
        wsmo#version hasValue "$Revision: 1.17 $"
    endAnnotations
    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"
    annotations
        dc#title hasValue "GGMediator 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/TR/d2/v1.2/#mediators"
        wsmo#version hasValue "$Revision: 1.17 $"
    endAnnotations
    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 fulfills at
* least partially the goal is explicitly stated in the wgMediator.
*/

```

```
wgMediator _"http://example.org/wgMediator"  
annotations  
  dc#type hasValue _"http://www.wsmo.org/2004/d2/v1.2/#mediators"  
endAnnotations  
source _"http://example.org/BirthRegistration"  
target _"http://example.org/GetCitizenship"
```

Appendix B. Datatypes and Built-ins in WSML

This appendix contains a preliminary list of built-in functions and relations for datatypes in WSML. It also contains a translation of syntactic shortcuts to datatype predicates.

Appendix B.1. WSML Datatypes

WSML recommends the use of XML Schema datatypes as defined in [Biron & Malhotra, 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 requires all compliant implementations to implement at least the string, decimal, and integer data types.

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 B.1 lists the datatypes recommended by WSML with the shortcut 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 & Malhotra, 2004]) and an example of the use of the datatype. The full name of the datatype constructor is the IRI <http://www.w3.org/2001/XMLSchema#datatype>, where *datatype* is the name of the XML schema datatype, depicted in Table B.1. The datatype `rdf#XMLLiteral` is defined in [Klyne & Carroll, 2004].

Datatype	Description of the Datatype	Syntax	Datatype constructor shortcut syntax
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: \\.	xsd:string("any-character*")	_string
decimal	That subset of natural numbers which can be represented with a finite sequence of decimal numerals.	xsd:decimal("-?numeric+.numeric+")	_decimal
integer	That subset of decimals which corresponds with natural numbers.	xsd:integer("-?numeric+")	_integer

Table B.1: WSML Datatypes

Datatype	Description of the Datatype	Syntax	Datatype constructor shortcut syntax
float		xsd#float("see XML Schema document")	_float
double		xsd#double("see XML Schema document")	_double
boolean		xsd#boolean("true-or-false")	_boolean
dateTime		xsd#dateTime(<i>integer_year</i> , <i>integer_month</i> , <i>integer_day</i> , <i>integer_hour</i> , <i>integer_minute</i> , <i>decimal_second</i> , <i>integer_timezone-hour</i> , <i>integer_timezone-minute</i>) xsd#dateTime(<i>integer_year</i> , <i>integer_month</i> , <i>integer_day</i> , <i>integer_hour</i> , <i>integer_minute</i> , <i>decimal_second</i>)	_dateTime
time		xsd#time(<i>integer_hour</i> , <i>integer_minute</i> , <i>decimal_second</i> , <i>integer_timezone-hour</i> , <i>integer_timezone-minute</i>) xsd#time(<i>integer_hour</i> , <i>integer_minute</i> , <i>decimal_second</i>)	_time
date		xsd#date(<i>integer_year</i> , <i>integer_month</i> , <i>integer_day</i> , <i>integer_timezone-hour</i> , <i>integer_timezone-minute</i>) xsd#date(<i>integer_year</i> , <i>integer_month</i> , <i>integer_day</i>)	_date
gYearMonth		xsd#gYearMonth(<i>integer_year</i> , <i>integer_month</i>)	_gyearmonth
gYear		xsd#gYear(<i>integer_year</i>)	_gyear
gMonthDay		xsd#gMonthDay(<i>integer_month</i> , <i>integer_day</i>)	_gmonthday
gDay		xsd#gDay(<i>integer_day</i>)	_gday
gMonth		xsd#gMonth(<i>integer_month</i>)	_gmonth
hexBinary		xsd#hexBinary("hexadecimal-encoding")	_hexbinary
base64Binary		xsd#base64Binary("base64-encoding")	_base64binary

Datatype	Description of the Datatype	Syntax	Datatype constructor shortcut syntax
XMLLiteral	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: \\, such that removing escaping yields a string in the lexical space of the rdf:XMLLiteral , specified in [Klyne & Carroll, 2004] .	<code>rdf#XMLLiteral(""-escaped XML content")</code>	

Table B.2 contains the shortcut syntax for the string, integer, and decimal datatypes.

Datatype constructor	Shortcut syntax	Example
<code>xsd#string</code>	<code>"any-Unicode-character*"</code>	"John Smith"
<code>xsd#decimal</code>	<code>'-'?numeric+.numeric+</code>	4.2, 42.0
<code>xsd#integer</code>	<code>'-'?numeric+</code>	42, -4

Table B.2: WSML Datatype shortcut syntax

Appendix B.2. WSML Built-in Predicates

This section contains a list of built-in predicates suggested for use in WSML. These predicates largely correspond to functions and comparators in XQuery/XPath [\[Malhotra et al., 2004\]](#). Notice that SWRL [\[Horrocks et al., 2004\]](#) built-ins are 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. A translation of the built-in symbols to datatype predicates is given 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 B.3 for the complete list. In case the built-in predicate has an equivalent function in XQuery, this is indicated in the table. The types 'wsml#equal', 'wsml#inequal' and 'wsml#strongEqual' refer to the unification operator, inequality and user-defined inequality, respectively.

WSML built-in predicate	XQuery function	Type (A)	Type (B)	Return type
wsml#equal		abstract	abstract	
wsml#inequal		abstract	abstract	
wsml#numericEqual(A,B)	op#numeric-equal(A,B)	numeric	numeric	
wsml#numericInequal(A,B)	op#boolean-equal(op#numeric-equal(A,B), "false")	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#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
wsml#stringEqual(A,B)	fn#codepoint-equal(A, B)	string	string	
wsml#stringInequal(A,B)	op#boolean-equal(fn:codepoint-equal(A, B), "false")	string	string	
wsml#dateEqual(A,B)	op#date-equal(A, B)	date	date	
wsml#dateInequal(A,B)	op#boolean-equal(op:date-equal(A, B), "false")	date	date	
wsml#dateGreaterThan(A,B)	op#date-greater-than(A, B)	date	date	
wsml#dateLessThan(A,B)	op#date-less-than(A, B)	date	date	
wsml#timeEqual(A,B)	op:date-equal(A, B)	time	time	
wsml#timeInequal(A,B)	op#boolean-equal(op:date-equal(A, B), "false")	time	time	
wsml#timeGreaterThan(A,B)	op#time-greater-than(A, B)	time	time	
wsml#timeLessThan(A,B)	op#time-less-than(A, B)	time	time	
wsml#dateTimeEqual(A,B)	op#dateTime-equal(A, B)	dateTime	dateTime	
wsml#dateTimeInequal(A,B)	op#boolean-equal(op:dateTime-equal(A, B), "false")	dateTime	dateTime	
wsml#dateTimeGreaterThan(A,B)	op#dateTime-greater-than(A, B)	dateTime	dateTime	
wsml#dateTimeLessThan(A,B)	op#dateTime-less-than(A, B)	dateTime	dateTime	

Table B.3: WSML Built-in Predicates

WSML built-in predicate	XQuery function	Type (A)	Type (B)	Return type
wsml#gyearmonthEqual(A,B)	op#gYearMonth-equal(A, B)	gYearMonth	gYearMonth	
wsml#gyearEqual(A,B)	op#gYear-equal(A, B)	gYear	gYear	
wsml#gmonthdayEqual(A,B)	op#gMonthDay-equal(A, B)	gMonthDay	gMonthDay	
wsml#gmonthEqual(A,B)	op#gMonth-equal(A, B)	gMonth	gMonth	
wsml#gdayEqual(A,B)	op#gDay-equal(A, B)	gDay	gDay	
wsml#durationEqual(A,B)	op#duration-equal(A, B)	duration	duration	
wsml#dayTimeDurationGreaterThan(A,B)	op#dayTimeDuration-greater-than(A, B)	dayTime	dayTime	
wsml#dayTimeDurationLessThan(A,B)	op#dayTimeDuration-less-than(A, B)	dayTimeDuration	dayTimeDuration	
wsml#yearMonthDurationGreaterThan(A,B)	op#yearMonthDuration-greater-than(A, B)	yearMonthDuration	yearMonthDuration	
wsml#yearMonthDurationLessThan(A,B)	op#yearMonthDuration-less-than(A, B)	yearMonthDuration	yearMonthDuration	

Each WSML implementation is required to either implement the complement of each of these built-ins or to provide a negation operator which can be used together with these predicates, with the following exceptions:

- wsml#equal and wsml#inequal are not required for WSML-Core

Appendix B.3. Translating Built-in Symbols to Predicates

In this section, we provide the translation of the built-in (function and predicate) symbols to 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	Type (A)	Type (B)	Predicate
A = B	abstract	abstract	wsml#equal(A,B)
A != B	abstract	abstract	wsml#inequal(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)

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

Operator	Type (A)	Type (B)	Predicate
A <= B	numeric	numeric	wsml#lessEqual(A,B)
A > B	numeric	numeric	wsml#greaterThan(A,B)
A >= B	numeric	numeric	wsml#greaterEqual(A,B)
A = B	string	string	wsml#stringEqual(A,B)
A != B	string	string	wsml#stringInequal(A,B)
A = B	date	date	wsml#dateEqual(A,B)
A != B	date	date	wsml#dateInequal(A,B)
A = B	time	time	wsml#timeEqual(A,B)
A != B	time	time	wsml#timeInequal(A,B)
A = B	dateTime	dateTime	wsml#dateTimeEqual(A,B)
A != B	dateTime	dateTime	wsml#dateTimeInequal(A,B)
A = B	gYearMonth	gYearMonth	wsml#gyearmonthEqual(A,B)
A = B	gYear	gYear	wsml#gyearEqual(A,B)
A = B	gMonthDay	gMonthDay	wsml#gmonthdayEqual(A,B)
A = B	gMonth	gMonth	wsml#gmonthEqual(A,B)
A = B	gDay	gDay	wsml#gdayEqual(A,B)
A = B	duration	duration	wsml#durationEqual(A,B)
A = B	dayTimeDuration	dayTimeDuration	wsml#dayTimeDurationEqual(A,B)
A = B	yearMonthDuration	yearMonthDuration	wsml#yearMonthDurationEqual(A,B)

We list the built-in functions and their translation to datatype predicates in Table B.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 B.5: Translation of WSMML infix operators to datatype predicates

Function symbols in WSMML 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 symbol 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:

$$?w = ?x1 + ?z \text{ and } \text{wsml\#numericAdd}(?x1, ?x, ?y)$$

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

$$?w = ?x2 \text{ and } \text{wsml\#numericAdd}(?x1, ?x, ?y) \text{ and } \text{wsml\#numericAdd}(?x2, ?x1, ?z)$$

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:

$$\text{wsml\#numericEqual}(?w, ?x2) \text{ and } \text{wsml\#numericAdd}(?x1, ?x, ?y) \text{ and } \text{wsml\#numericAdd}(?x2, ?x1, ?z)$$

Appendix C. WSML Keywords

This appendix lists all WSML keywords, along 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 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 C.1.

Keyword	Section
<u>wsmIVariant</u>	<u>2.2.1</u>
<u>namespace</u>	<u>2.2.2</u>
<u>annotations</u>	<u>2.2.3</u>
<u>endAnnotations</u>	<u>2.2.3</u>
<u>importsOntology</u>	<u>2.2.3</u>
<u>usesMediator</u>	<u>2.2.3</u>
<u>ontology</u>	<u>2.3</u>
<u>concept</u>	<u>2.3.1</u>
<u>subConceptOf</u>	<u>2.3.1</u>
<u>ofType</u>	<u>2.3.1</u>
<u>impliesType</u>	<u>2.3.1</u>
<u>transitive</u>	<u>2.3.1</u>

Table C.1: WSML keywords

Keyword	Section
<u>symmetric</u>	<u>2.3.1</u>
<u>inverseOf</u>	<u>2.3.1</u>
<u>subAttributeOf</u>	<u>2.3.1</u>
<u>reflexive</u>	<u>2.3.1</u>
<u>instance</u>	<u>2.3.2</u>
<u>memberOf</u>	<u>2.3.2</u>
<u>hasValue</u>	<u>2.3.2</u>
<u>relation</u>	<u>2.3.3</u>
<u>subRelationOf</u>	<u>2.3.3</u>
<u>relationInstance</u>	<u>2.3.4</u>
<u>axiom</u>	<u>2.3.5</u>
<u>definedBy</u>	<u>2.3.5</u>
<u>capability</u>	<u>2.4.1</u>
<u>sharedVariables</u>	<u>2.4.1</u>
<u>precondition</u>	<u>2.4.1</u>
<u>assumption</u>	<u>2.4.1</u>
<u>postcondition</u>	<u>2.4.1</u>
<u>effect</u>	<u>2.4.1</u>
<u>interface</u>	<u>2.4.2</u>
<u>choreography</u>	<u>2.4.2</u>
<u>stateSignature</u>	<u>2.4.2</u>
<u>static</u>	<u>2.4.2</u>
<u>in</u>	<u>2.4.2</u>
<u>out</u>	<u>2.4.2</u>
<u>shared</u>	<u>2.4.2</u>
<u>controlled</u>	<u>2.4.2</u>
<u>withGrounding</u>	<u>2.4.2</u>
<u>if</u>	<u>2.4.2</u>

Keyword	Section
<u>endif</u>	<u>2.4.2</u>
<u>forall</u>	<u>2.4.2</u>
<u>endForall</u>	<u>2.4.2</u>
<u>with</u>	<u>2.4.2</u>
<u>do</u>	<u>2.4.2</u>
<u>choose</u>	<u>2.4.2</u>
<u>endChoose</u>	<u>2.4.2</u>
<u>add</u>	<u>2.4.2</u>
<u>delete</u>	<u>2.4.2</u>
<u>update</u>	<u>2.4.2</u>
<u>orchestration</u>	<u>2.4.2</u>
<u>nonFunctionalProperty</u>	<u>2.4.3</u>
<u>nfp</u>	<u>2.4.3</u>
<u>goal</u>	<u>2.5</u>
<u>ooMediator</u>	<u>2.6</u>
<u>ggMediator</u>	<u>2.6</u>
<u>wgMediator</u>	<u>2.6</u>
<u>wwMediator</u>	<u>2.6</u>
<u>source</u>	<u>2.6</u>
<u>target</u>	<u>2.6</u>
<u>usesService</u>	<u>2.6</u>
<u>webService</u>	<u>2.7</u>

Table C.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 B](#)

Keyword	WSML-Core	WSML-DL	WSML-Flight	WSML-Rule	WSML-Full
true	+	+	+	+	+
false	+	+	+	+	+

Table C.2: WSML logical expression keywords

Keyword	WSML-Core	WSML-DL	WSML-Flight	WSML-Rule	WSML-Full
memberOf	+	+	+	+	+
hasValue	+	+	+	+	+
subConceptOf	+	+	+	+	+
ofType	+	+	+	+	+
impliesType	+	+	+	+	+
and	+	+	+	+	+
or	+	+	+	+	+
implies	+	+	+	+	+
impliedBy	+	+	+	+	+
equivalent	+	+	+	+	+
neg	-	+	-	-	+
naf	-	-	+	+	+
forall	+	+	+	+	+
exists	-	+	-	-	+

Appendix D. Changelog

In the following we will outline changes corresponding to the development of WSML v0.21 to WSML v1.0 and this document as a whole, including:

- Items in WSML v1.0 that are not backward compatible to previous WSML versions.
- All changes in the language from WSML v0.21 to WSML v1.0
- Changes in this document since the latest version

D.1. Non backward compatible items in WSML v1.0

The following list describes all items in the language that cause WSML v1.0 to not be backward compatible to the previous versions of WSML:

- The syntax used to specify annotations and non functional properties. The non functional properties of previous WSML versions have been replaced with the notion of annotations. Annotations can belong to either a whole WSML document or to any single WSML element. They can (typically) contain attributes belonging to, e.g., the Dublin Core Metadata Element Set, but are not restricted to.
A new notion of non-functional property has been introduced in WSML v1.0 to describe all those properties which strictly belong to a Web service, goal, capability, interface or mediator and which are not functional and behavioral.
For more details on the specification of annotations or non functional properties please see the sections [2.2.3 Header \(Annotations\)](#) and [2.4.3 Non-functional Properties](#).
- The usage of relations is not any longer allowed in WSML-Core and WSML-DL.
- The symbol `:=`: (strong equality) has been removed.
- The datatypes `_iri` and `_sqname` have been removed. IRIs are abstract symbols interpreted arbitrarily, in contrast to data values which are concrete symbols interpreted according to a particular datatype. Depicting an IRI as a datatype has been misleading in this respect.
- The keyword 'comment' has been removed from the language and can no longer be used to write single line comments.

D.2. Changes in the language (WSML v1.0) since version 0.21

- The concept of non-functional properties in version 0.21 has been split into two notions: [annotations](#) and [nonfunctional properties](#) (see also D.1 above).
- The new non functional properties are not written as blocks anymore, but one by one, using the `nonFunctionalProperty` or `nfp` keywords. In the scope of the new non functional properties variables have been allowed as new type of identifiers.
- The usage of relations is not any longer allowed in WSML-Core and WSML-DL (see also D.1 above).

- Equality and negated equality ($a=b$, $\text{neg } a=b$) have been allowed in WSMML-DL logical expressions for instances.
- The symbol $:=$: (strong equality) has been removed (see also D.1 above).
- The datatypes `_iri` and `_sname` have been removed (see also D.1 above).
- Datatype wrappers are now IRIs based on the XML Schema datatypes (e.g. `xsd#date(...)`); the names of datatype constructors as used in the previous version (e.g. `_date(...)`) may still be used as shortcuts, but their usage is discouraged.
- The new datatype `rdf#XMLLiteral` is introduced.
- `sQNames` have been renamed to Compact URIs.
- A syntax for choreographies, based on WSMO deliverable 14 is now included in the language.
- `wgMediators` allow multiple sources.
- It is no longer allowed to use an anonymous identifier for the identification of the following elements: ontology, goal, Web service, mediator, capability, interface, choreography and orchestration.
- Additionally to the sets of identifiers of ontology, goal, Web service, `ooMediator`, `ggMediator`, `wgMediator` and `wwMediator`, the sets of identifiers of the following items are now as well disjoint: capability, interface, choreography, orchestration, state signature, grounding identifier, variant identifier, datatype wrapper identifier, built-in predicate identifier.
- Due to (a) the fact that (as described in the previous item) the elements ontology, goal, Web service, mediator, capability, interface, choreography and orchestration need to have disjoint identifiers, and (b) the fact that if no identifier is specified for those elements, the identifier is assumed to be the same as the locator of the specification, it is not allowed to write two such elements (as top level entities) in a single file. It is nevertheless possible to write nested elements without identifier in one single file: nested capabilities, interfaces, choreographies and orchestrations do not need to have an identifier at all.
- Imported ontologies can be WSMML, RDFS, or OWL DL ontologies.
- The keyword 'comment' has been removed (see also D.1 above).

D.3. Changes in this document since the previous version (July 28, 2008)

- Made examples throughout the deliverable consistent.
- Added last updated WSMML Grammar.
- Corrected structure in WSMML-Core Section 3.
- Restructured the order of Goal, Web Service and Mediators.
- The identifiers for 'useService', 'target' and 'source' for mediators are not optional.
- Fixed all grammar excerpts in the syntax part of the deliverable and made them consistent with the grammar in the Appendix.
- Moved the grammar excerpts and general textual parts on identifiers from the Anonymous Identifiers subsection to the general section on identifiers at the beginning of Section 2.1.2.
- Corrected text on state signatures and rules in Choreography Section.

D.4. Issues List

For a list of the issues to be addressed in this version of D16.1, see the [WSML Issues List](#).

References

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

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

[Boley & Kifer, 2008] H. Boley & M. Kifer, (Eds.). *RIF Basic Logic Dialect*, W3C Working Draft. Available from <http://www.w3.org/TR/rif-blld/>.

[Borgida, 1996] A. Borgida. On the Relative Expressiveness of Description Logics and Predicate Logics. *Journal of Artificial Intelligence*, 82 (1-2): 353-367, 1996.

[Bray et al., 2004] 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/>.

[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, 2005] J. de Bruijn (Ed). *WSML Reasoner Implementation*. WSML Working Draft D16.2v0.2, 2005. 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.

[de Bruijn & Heymans, 2007] J. de Bruijn and S. Heymans. *WSML Ontology Semantics*. WSML Final Draft D28.3v0.2, 2007. Available from <http://www.wsmo.org/TR/d28/d28.3/v0.2/20070416/>.

[de Bruijn, 2008a] J. de Bruijn (Ed). *RIF RDF and OWL Compatibility*, W3C Working Draft. Available from <http://www.w3.org/TR/rif-rdf-owl/>.

[de Bruijn, 2008b] J. de Bruijn (Ed). *WSML/RDF*. WSML Working Draft D32v0.2, 2008. Available from <http://www.wsmo.org/TR/d32/v0.2/>.

[de Bruijn, 2008c] J. de Bruijn (Ed). *WSML Abstract Syntax*. WSML Working Draft D16.3v0.3, 2008. Available from <http://www.wsmo.org/TR/d16.3/v0.3/>.

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

[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., 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.

[Gelfond & Lifschitz, 1988] Gelfond and Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming (ICLP1988)*. MIT Press, Seattle, Washington, 1070-1080.

[Grosf et al., 2003] B. N. Grosf, 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. Grosf 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., 2005] U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel: Automatic location of services. In *Proceedings of the 2nd European Semantic Web Conference (ESWC2005)*, pp. 1-16. Springer-Verlag, 2005.

[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.

[Klyne & Carroll, 2004] G. Klyne, J. J. Carroll, (Eds.). *Resource Description Framework (RDF): Concepts and Abstract Syntax*, W3C Recommendation, 10 February 2004. Available from <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.

[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/>.

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

[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., 2006] D. Roman, H. Lausen, and U. Keller (eds.): *Web Service Modeling Ontology (WSMO)*, WSMO deliverable D2v1.3. available from <http://www.wsmo.org/TR/d2/v1.3/>.

[Scicluna et al., 2005] J. Scicluna, A. Polleres, D. Roman, and C. Feier (eds.): *Ontology-based Choreography and Orchestration of WSMO Services*, WSMO deliverable D14 version 0.2. available from <http://www.wsmo.org/TR/d14/v0.2/>.

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

[Steinmetz, 2008] Nathalie Steinmetz (Ed).: *WSML/OWL Mapping*, WSML Working draft D37 version 0.1 available from <http://www.wsmo.org/TR/d37/v0.1/>.

[Toma, 2008] Ioan Toma (Ed).: *WSML/XML*, WSML Working draft D36 version 0.1 available from <http://www.wsmo.org/TR/d36/v0.1/>.

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

[Yang & Kifer, 2002] G. Yang and M. Kifer. *Well-Founded Optimism: Inheritance in Frame-Based Knowledge Bases*. In First International Conference on Ontologies, Databases and Applications of Semantics (ODBASE), Irvine, California, 2002.

[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.

Acknowledgements

The work is funded by the European Commission under the projects SWING, SOA4All and Service Web 3.0.

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 WSMML-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) rather than through their common core described in WSMML-Core. If we chose 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.