D16.1v0.3 WSML Language Reference

WSML Working Draft 18 March 2008

This version
http://www.wsmo.org/TR/d16/d16.1/v0.3/20080318/
Latest version
http://www.wsmo.org/TR/d16/d16.1/v0.3/
Previous version
http://www.wsmo.org/TR/d16/d16.1/v0.3/20080226/

Editors:
Ioan Toma
Nathalie Steinmetz
Authors:
The WSML working group members

Copyright © 2008 by the authors.

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 WSML specification is not backward compatible with respect to the syntax used to specify annotations descriptions. For more details please see Section 2.2.3

Table of Contents

PART I: PRELUDE

1. Introduction

- 1.1. Structure of the deliverable

PART II: WSML VARIANTS

2 WSML Syntax

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

3 WSML-Core

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

4 WSML-DL

- 4.1. 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

PART III: THE WSML EXCHANGE SYNTAXES

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

PART I: PRELUDE

1. Introduction

The Web Service Modeling Ontology WSMO [Roman *et al.* 2005] 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, 2007] 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 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. To increase the interoperability of WSML, exchange syntaxes to XML and RDFS, as well as a mapping to OWL DL have been developed (based on [de Bruijn, 2007]) in the following documents: WSML/XML in [Toma, 2008], WSML/RDF in [de Bruijn, 2006] and the WSML/OWL mapping in [Steinmetz, 2008].

That is in this document, the WSML Language Reference, we take the abstract syntax of WSML as a basis for the specification of a family of Web Service description and Ontology specification languages. The Web Service Modeling Language (WSML) aims at providing means to formally describe all the elements defined in WSMO. The different variants of WSML correspond with different levels of logical expressiveness and the use of different languages paradigms. More specifically, we take Description Logics, First-Order Logic and Logic Programming as starting points for the development of the WSML language variants. The WSML language variants are both syntactically and semantically layered.

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

1.1. Structure of the Document

The remainder of this document is structured as follows:

PART II: WSML VARIANTS

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

PART III: THE WSML EXCHANGE SYNTAXES

The WSML variants are described in terms of their normative human-readable language in PART II. Although this syntax has been formally specified in [de Bruijn, 2007], as well as in the form of a grammar (see also Appendix A), there are limitations with respect to the exchange of the syntax over the Web. Therefore three syntaxes for WSML were developed, namely XML, RDF and OWL. The XML exchange syntax for WSML, defined in [Toma, 2008], is the preferred syntax for the exchange of WSML specifications between machines. The RDF syntax of WSML, defined in [de Bruijn, 2006], can be used by RDF-based applications and finally a mapping between WSML and OWL ontologies is discussed in [de Bruijn, 2007] and in [Steinmetz, 2008] in order to allow interoperation with OWL-based applications. This section provides pointers to external documents containing these syntaxes.

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.

Please note that the current version of WSML specification is not backward compatible with respect to the syntax used to specify annotations descriptions. For more details please see Section 2.2.3

PART II: WSML VARIANTS

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

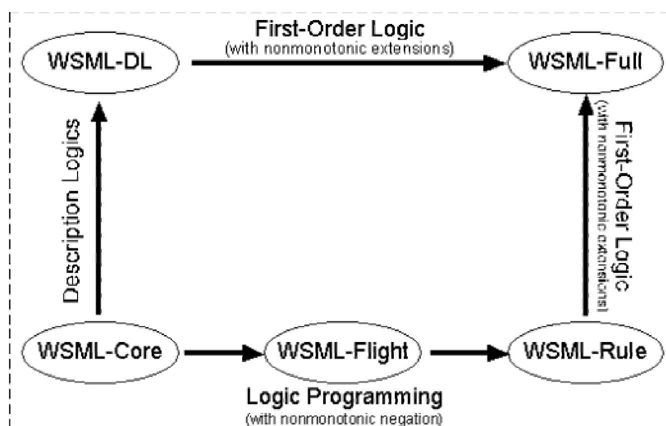

Figure 1. WSML Space

WSML-Core

This language is defined by the intersection of Description Logic and Horn Logic, based on Description Logic Programs [Grosof et al., 2003]. It has the least expressive power of all the languages of the WSML family and therefore has the most preferable computational characteristics. The main features of the language are the support for modeling classes, attributes, binary relations and instances. Furthermore, the language supports class hierarchies, as well as relation hierarchies. WSML-Core provides support for datatypes and datatype predicates.[2] WSML-Core is based on the intersection of Description Logics and Datalog, corresponding to the DLP fragment [Grosof et al., 2003].

WSML-DL

This language is an extension of WSML-Core which fully captures the Description Logic *SHIQ*(D), which captures a major part of the (DL species of the) Web Ontology Language OWL [Dean & Schreiber, 2004]

WSML-Flight

This language is an extension of WSML-Core with such features as meta-modeling, constraints and nonmonotonic negation. WSML-Flight is based on a logic programming variant of F-Logic [Kifer et al., 1995] and is semantically equivalent to Datalog with inequality and (locally) stratified negation. As such, WSML-Flight provides a powerful rule language.

WSML-Rule

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

WSML-Full

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

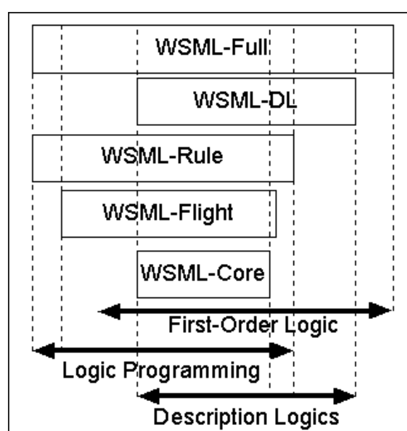

Figure 2. WSML Layering

As can be seen from Figure 2, WSML has two alternative layerings, namely, WSML-Core -> WSML-DL -> WSML-Full and WSML-Core -> WSML-Flight -> WSML-Rule -> WSML-Full. In both layerings, WSML-Core is the least expressive and WSML-Full is the most expressive language. The two layerings are to a certain extent disjoint in the sense that interoperation between the Description Logic variant (WSML-DL) on the one hand and the Logic Programming variants (WSML-Flight and WSML-Rule) on the other, is only possible through a common core (WSML-Core) or through a very expressive (undecidable) superset (WSML-Full). However, there are proposals which allow interoperation between the two while retaining decidability of the satisfiability problem, either by reducing the expressiveness of 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 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.

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 are described in Section 2.2. WSML ontologies are described in Section 2.3. The elements shared between goals and web services, namely, capabilities and interfaces, are described in Section 2.4. Goals, mediators and web services are described in Sections 2.5, 2.6 and 2.7, respectively. Finally, the WSML logical expression syntax is specified in Section 2.8.

2.1 WSML Syntax Basics

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

Nonetheless, attribute names are global and it is possible to specify global behavior of attributes through logical expressions. However, we do not expect this to be necessary in the general case and we strongly advise against it. 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 identifiers have been introduced, namely variables which might be used in the conceptual syntax only in **nonFunctionalProperties** block definition or in the **sharedVariables** block.

Data values

WSML 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, WSML 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 WSML 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 WSML implementation is required to support at least the *string*, *integer* and *decimal* datatypes.

The WSML surface syntax allows, but discourages, to use 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 WSML 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
```

WSML allows the following syntactical shortcuts for particular datatypes:

- Data values of type *string* can be written between double quotes “”. Double quotes inside a string should be escaped using the backslash (‘\’) 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 decimal symbols. 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 WSML 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. an *integer*, a *decimal* or a *string*, or it can be a constructed datavalue, where a *constructeddatavalue* corresponds to a datatype wrapper.

```
datavalue           = simpledatavalue
                    | constructeddatavalue
datavaluelist       = simpledatavalue ( ',' simpledatavalue )*
simpledatavalue     = integer
                    | decimal
                    | string
constructeddatavalue = iri '(' datavaluelist ')'
```

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. WSML adopts a view similar to the RDF-like version of QNames, but due to its deviation from the original definition in XML we call them '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, except for attribute identifiers. The namespace of an attribute identifier with no prefix is the namespace of the concept type or the instance definition within which it appears. *Note:* In case the default namespace is not defined or a prefix used in a Compact URI cannot be resolved, the WSML 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 WSML keywords. The characters '.' and '-' in a Compact URI need to be escaped using the backslash ('\') character.

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

Examples of full IRIs in WSML:

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

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

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

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

Anonymous Identifiers

An anonymous identifier represents 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.

The use of an identifier in the specification of the WSML elements *ontology*, *goal*, *web service*, *mediator*, *capability*, *interface*, *import ontologies*, *used mediators*, *sources of mediators*, *targets of mediators*, *services used by mediators*, *choreography* and *orchestration* is optional. If no identifier is specified, 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.

Note that anonymous identifiers are disallowed for the top-level elements, namely *ontology*, *goal*, *webService*, *ooMediator*, *ggMediator*, *wgMediator* and *wwMediator*. Anonymous identifiers are as well disallowed for *capability*, *interface*, *import ontologies*, *used mediators*, *sources of mediators*, *targets of mediators*, *services used by mediators*, *choreography* and *orchestration*.

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

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 conjunctively. 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 conjunction of the ranges of the original

attributes.

Invalid 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 WSML document. Examples include Compact URIs with undefined namespace prefixes and invalid data values.

Variables

Variables are a new kind of identifiers which are used in defining WSML logical expressions (see Section Section 2.8).

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

Variables may only be used inside of **logical expressions**, as values in **non-functional properties** block definitions (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

2.1.3 Comments in WSML

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

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

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

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 Elements

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

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

2.2.1 WSML Variant

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

Table 2.1: WSML variant identifiers

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

The specification of the **wsmlVariant** is optional. In case no variant is specified, no guarantees can be made with respect to the specification and WSML-Full may be assumed.

wsmlvariant = 'wsmlVariant' full_iri

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

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

When the intended WSML 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 WSML specifications to stay within desired limits of complexity and to communicate their desires to others.

2.2.2 Namespace References

At the top of a WSML document, below the identification of the WSML variant, there is an optional block of namespace references, which is preceded by the **namespace** keyword. The **namespace** keyword is followed by a number of namespace references. Each namespace reference, except for the default namespace, consists of the chosen prefix and the IRI which identifies the namespace. Note that, like any argument list in WSML, the list of namespace references is delimited with curly brackets '{' '}'. In case only a default namespace is declared, the curly brackets are not required.

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

Two examples 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/",
     wsml _"http://www.wsmo.org/wsml-syntax#",
     loc _"http://www.wsmo.org/ontologies/location#",
     oo _"http://example.org/ooMediator#"}

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

2.2.3 Header

Any WSML specification may have annotations, may import ontologies and may use mediators:

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

Annotations

Annotations may be used for the WSML document as a whole but also for each element in the specification. Annotations 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, which may be any identifier and can thus be an IRI, a data value, an anonymous identifier or a comma-separated list of the former, delimited with curly brackets. The recommended properties are the properties of the Dublin Core Metadata Element Set [Weibel et al. 1998], but the list of properties is extensible and thus the user can choose to use properties coming from different sources. WSMO [Roman et al., 2005] defines a number of properties which are not in the Dublin Core. These properties can be used in a WSML specification by referring to the WSML namespace (http://www.wsmo.org/wsml/wsml-syntax#). These properties are: **wsml#version** and **wsml#owner**, (here we assume that the prefix wsml has been defined as referring to the WSML namespace; see Section 2.1.1). For recommended usage of annotations see [Roman et al., 2005]. Following the WSML convention, if a property has multiple values, these are separated by commas and the list of values is delimited by curly brackets.

annotations = 'annotations' attributevalue* 'endAnnotations'

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://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"
  wsml#version hasValue "$Revision: 1.224 $"
endAnnotations
```

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

The current version of WSML 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 introduced by *nonFunctionalProperties* or *nfp* constructs. The current versions use the *annotations* construct. A translation tool from the old syntax to the new syntax used to specify annotations is available at http://tools.deri.org/wsml/nfptranslator/.

Importing Ontologies

Ontologies may be imported in any WSML specification through the import ontologies block, identified by the keyword **importsOntology**. Following the keyword is a list of IRIs identifying the ontologies being imported. An **importsOntology** definition serves to merge ontologies, similar to the **owl:import** annotation property in OWL. This means the resulting ontology is the union of all axioms and definitions in the importing and imported ontologies. Please note that recursive import of ontologies is also supported. This means that if an imported ontology has any imported ontologies of its own, these ontologies are also imported.

WSML does not only allow the import of WSML ontologies: the imported ontologies may be WSML ontologies, RDFS graphs, or OWL ontologies (both OWL DL and OWL Full are supported). The variant of every imported ontology should be lower than or equal to the variant of the importing description. RDFS ontologies are conformant to WSML Flight or WSML Rule. If an OWL DL ontology is in the DLP subset, i.e. it is equivalent to a set of equality-free Horn formulas, it has the WSML-Core variant. If not, it has the variant WSML-DL or WSML-Full. An OWL Full ontology is in the WSML-Full variant.

If the imported ontology is of a higher WSML 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 WSML-DL ontology in a WSML-Rule specification), the resultant will be of the least common superset of the variants. In the case of WSML-DL and WSML-Rule, the least common superset is WSML-Full.

importsontology = 'importsOntology' irilist

Example:

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

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

Eexample:

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

2.3 Ontology Specification in WSML

A WSML ontology specification is identified by the **ontology** keyword optionally followed by an IRI which serves as the identifier of the ontology. 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 WSML language. The modeling elements are based on the WSMO conceptual model of ontologies [Roman et al., 2005].

2.3.1 Concepts

A concept definition starts with the **concept** keyword, which is 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 WSML 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
  hasParent impliesType Human
  hasChild impliesType Human
  hasAncestor impliesType Human
  hasRelative impliesType Human
  hasWeight ofType xsd#float
  hasWeightInKG ofType xsd#float
  hasBirthdate ofType xsd#date
  hasObit ofType xsd#date
  hasBirthplace ofType loc#location
  isMarriedTo impliesType Human
  hasCitizenship ofType oo#country
```

WSML allows 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 WSML 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 (called "complete classes" in OWL). The definition of a defined concept is not only necessary, but also sufficient. A necessary definition, such as the concept specification in the example above, specifies implications of membership of the concept for all instances of this concept. WSML 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, hasWeight etc. must be of specific types. A necessary and sufficient definition also works the other way around, which means that if certain properties hold for the instance, the instance is inferred to be 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 **memberOf** A **equivalent** ?x **memberOf** B_1 **and** ... **and** ?x **memberOf** B_n

With A and B_1,...,B_n concept identifiers.

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

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

Attributes

Two important features of attribute modeling which set WSML apart from OWL are cardinality constraints and attribute range constraints. OWL offers cardinality and range restrictions on attributes which serve to create additional (monotonic) inferences such as existence or equality of objects or membership in a certain class. For many users these restrictions show unintuitive behavior from the viewpoint of classical rule languages and databases [de Bruijn et al., 2005]: Means to actually check the data in your knowledge base with respect to integrity constraints are missing in OWL. WSML 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.

WSML 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 WSML variants, namely, WSML-Core and WSML-DL, do not allow constraining attribute definitions. In order to facilitate conceptual modeling in these language variants, we allow the use of **impliesType** in WSML.

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

The keyword **impliesType** can be used for inferring the type of a particular attribute value. An attribute definition of the form *A* **impliesType** *D*, where *A* is an attribute identifier and *D* is a concept identifier, implies membership of the concept *D* for all values of the attribute *A*. Please note that 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.

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.

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	=	[attr]: id attributefeature* att_type cardinality? [type]: idlist annotations?
att_type	=	'ofType'
	|	'impliesType'
cardinality	=	'(' [min_cardinality]: digit+ [max_cardinality]: cardinality_number? ')'
cardinality_number	=	{finite_cardinality} digit+
	|	{infinite_cardinality} '*'
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
  hasParent inverseOf(hasChild) impliesType Human
  hasChild subAttributeOf(hasRelative) impliesType Human
  hasAncestor transitive impliesType Human
  hasRelative symmetric impliesType Human
  hasWeight ofType (1) xsd#float
  hasWeightInKG ofType (1) xsd#float
  hasBirthdate ofType (1) xsd#date
  hasObit ofType (0 1) xsd#date
  hasBirthplace ofType (1) loc#location
  isMarriedTo symmetric impliesType (0 1) Human
  hasCitizenship ofType oo#country
```

2.3.2 Relations

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

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

relation	=	'relation' id arity? paramtyping? superrelation? annotations?
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 distance (ofType City, ofType City, impliesType _decimal) subRelationOf measurement

relation distance/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.

2.3.3 Instances

An instance definition starts with the **instance** keyword, followed by the identifier of the instance, the **memberOf** keyword and the name of the concept to which the instance belongs. The **memberOf** keyword identifies the concept to which the instance belongs. This definition is followed by the attribute values associated with the instance. Each property filler consists of the property identifier, the keyword **hasValue** and the value(s) for the attribute.

instance = 'instance' id memberof? 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., 2005], 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 WSML variants by using an empty idlist, e.g. 'a **ofType** { }'

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.

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

2.3.4 Axioms

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?
| id? annotations? log_definition
log_definition = 'definedBy' log_expr+

Example of a defining axiom:

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

WSML 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 Capability, Interface and Non-functional Properties Specification 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 optional **nonfunctionalproperties** . 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 WSML 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 WSML:

```
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, optional **nonfunctionalproperties** 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 in WSML is defined. 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 **usedMediator** 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.

interfaces	=	interface
		| minterfaces
minterfaces	=	'interface' '{' iri moreiris* '}'
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** 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.

```
statesignature  = 'stateSignature'  id?  header*  mode*
mode            = mode_id  mode_entry_list
mode_entry_list = {mode_entry}       mode_entry
                | {mode_entry_list} mode_entry','  mode_entry_list
mode_id         = {static}      'static'
                | {in}          'in'
                | {out}         'out'
                | {shared}      'shared'
                | {controlled} 'controlled'
mode_entry      = {default_mode}  iri  grounding?
                | {concept_mode} 'concept'  iri  grounding?
                | {relation_mode} 'relation'  iri  grounding?
grounding       = 'withGrounding'  grounding_info
grounding_info  = {iri}     iri
                | {listiri} '{'  listiri'}'
listiri         = {iri}     iri
                | {listiri} iri','  listiri
```

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, a set of piped-rules (for non-determinism) 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 ending with 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 ending with 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 ends with the **endChoose** keyword. A piped rule is either a normal rule element or defined recursively by a rule followed by a pipe or followed by another set of piped rules.

```
transitions  = 'transitionRules'  id?  annotations?  rule*
rule         = {if}        'if'  condition  'then'  rule+  'endIf'
             | {forall}    'forall'  variablelist  'with'  condition  'do'  rule+  'endForall'
             | {choose} 'choose'  variablelist  'with'  condition  'do'  rule+  'endChoose'
             | {uncond} piped_rules
             | {update} updaterule
condition    = {restricted_le} expr
piped_rules  = {rule}   rule
             | {piped} rule'|'  piped_rules
```

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}      'add'
               | {delete}  'delete'
               | {update} 'update'
fact           = {fact_preferred}       term  attr_fact?  'memberOf'  termlist  fact_update?
               | {fact_nonpreferred} term  'memberOf'  termlist  fact_update?  attr_fact
               | {fact_molecule}        term  attr_fact
               | {fact_relation}        '@'  id  '('term_updates')'
fact_update    = '=>'  termlist
attr_fact      = '['  attr_fact_list']'
attr_fact_list = {attr_relation} term'hasValue'  termlist  fact_update?
               |                 attr_fact_list','  term'hasValue'  termlist  fact_update?
term_updates   = {one_param}     term_update
               | {more_params} term_update','  term_updates
term_update    = {single} term
               | {move} [oldterm]:  term  '=>'  [newterm]:  term
```

new_term = {new_term} '=>' term

The syntax presented before, known more commonly as the WSML Abstract State Machine (ASM) syntax, is the only syntax for choreographies that may be embedded within a WSML 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 WSML group it is currently not possible to embed an orchestration within a WSML 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(MediaShoppingServicePortType/SearchCatalog/In)",

   out
    mediaproduct#MediaProduct withGrounding
    _"http://example.org/webservices/shopping/mediashoppingservice#wsdl.interfaceMessageReference(MediaShoppingServicePortType/SearchCatalog/Out)",

  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 WSML 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 or a variable symbol (or a set of such symbols). This is followed by an optional **annotations** block.

Please 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 WSML 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
      | 'nonFunctionalProperties'  attributevaluenfp
attributevaluenfp =  id  'hasValue'  valuelistnfp  annotations?  log_definition?
```

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

```
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#YearsDuration]
          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 Goal Specification in WSML

A WSML goal specification is identified by the **goal** keyword optionally followed by an IRI which serves as the identifier of the goal. 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?  interfaces*
```

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 Mediator Specification in WSML

WSML allows for the specification of four kinds of mediators, namely ontology mediators, mediators between Web services, mediators between goals and mediators between Web services and goals. These mediators are referred via the keywords **ooMediator**, **wwMediator**, **ggMediator** and **wgMediator**, respectively (cf. [Roman et al., 2005]).

```
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?  (  ','  iri  )*  '}'
sources     =  source
             | msources
target      =  'target'  iri?
```

2.6.1 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 WSML may only contain identifiers of ontologies and other ooMediators as source.

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

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

oomediator = 'ooMediator' iri? annotations? importsOntology? nfp? sources target? use_service?

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

In fact, importing ontologies can be seen as a simple form of mediation, in which no heterogeneity is resolved. However, 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.

2.6.2 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?

2.6.3 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?

2.6.4 wgMediators

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

wgMediators in WSML may only have one **source**. The source may be the 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? source? target? use_service?

By externalizing the mediation services from the implementation of ontologies, goals and Web services, WSML allows loose coupling of elements; the mediator is responsible for relating the different elements to each other and resolving conflicts and mismatches. For more details we refer to [Roman et al., 2005].

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.

2.7 Web Service Specification in WSML

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

A Web service specification document in WSML consists of:

webservice = 'webService' 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.

2.8 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 V_{ID}.
- A set of function symbols V_F V_O.
- A set of datatype wrappers V_D V_O.
- A set of data values V_{DV} V_O which encompasses all string, integer and decimal values.
- A set of anonymous identifiers V_A V_O of the form _#, _#1, _#2, etc....
- A set of relation identifiers V_R V_{ID}.
- A set of variable identifiers V_V 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 V_O is a term.
- Any v V_V is a term
- If f V_F and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.
- If f V_D and $dv_1, ..., dv_n$ are in V_{DV} V_V, then $f(dv_1, ..., dv_n)$ is a term.

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

Based on the basic constructs of logical expressions, the terms, we can now define formulae. In WSML, we have atomic formulae and complex formulae. 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 *α, β* *Term(V)* and *γ* *Term(V)* or *γ* is of the form { $\gamma_1,...,\gamma_n$ }, with $\gamma_1,...,\gamma_n$ *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).
  - *α*[*β* **ofType** *γ*] is an atomic formula in *L*(*V*). Here, α identifies a concept, β identifies an attribute and γ identifies (a) concept (s).
  - *α*[*β* **impliesType** *γ*] is an atomic formula in *L*(*V*). Here, α identifies a concept, β identifies an attribute and γ identifies (a) concept(s).
  - *α*[*β* **hasValue** *γ*] 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 V_R and $t_1, ..., t_n$ are terms, then $r(t_1, ..., t_n)$ is an atomic formula in *L*(*V*).
- If *α*, *β* *Term(V)* then *α* **=** *β* and *α* **!=** *β* 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 $?x_1,...,?x_n$ be variables, then:
  - *α* **and** *β* is a formula in *L*(*V*).
  - *α* **or** *β* is a formula in *L*(*V*).
  - **neg** *α* is a formula in *L*(*V*).
  - **naf** *α* is a formula in *L*(*V*).
  - **forall** $?x_1,...,?x_n$ (*α*) is a formula in *L*(*V*).
  - **exists** $?x_1,...,?x_n$ (*α*) is a formula in *L*(*V*).
  - *α* **implies** *β* is a formula in *L*(*V*).
  - *α* **impliedBy** *β* is a formula in *L*(*V*).
  - *α* **equivalent** *β* is a formula in *L*(*V*).
  - *α* **:-** *β* is a formula in *L*(*V*). This formula is called an *LP (Logic Programming) rule*. *α* is called the *head* and *β* is called the *body* of the rule.
  - **!-** *α* is a formula in *L*(*V*). This formula is called a *constraint*. We say *α* is a constraint of the knowledge base.

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

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

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 [Grosof et al., 2003]. More specifically, WSML-Core is based on plain (function- and negation-free) Datalog, thus, the decidability and complexity results of Datalog apply to WSML-Core as well. The most important result is that Datalog is data complete for P, which means that query answering can be done in polynomial time.[2]

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

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

3.1 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. [Grosof et al., 2003].

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

3.2.1 Concepts

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

WSML-Core does not allow for the specification of the attribute features **reflexive**, **transitive**, **symmetric**, **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 inversity of attributes just like in DLs or OWL by defining respective axioms (cf. Definition 3.3 below).

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

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

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

3.2.2 Relations

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

3.2.3 Instances

In WSML-Core, nested molecules in attributes values are not allowed. The only allowed values as attribute values are strings, numbers, ids, or a lists of such values.

```
instance          = 'instance' id? memberof? annotations? attributevalue*
memberof          = 'memberOf' idlist
attributevalue    = id 'hasValue' valuelistnonest
valuelistnonest   = instattvalue
                  | '{' instattvalue moreinstattvalues* '}'
```

instattvalue = id
| number
| string

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

3.2.4 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. Mediators in WSML-Core

Mediators in WSML-Core follow the common WSML syntax.

3.5. Web Services in WSML-Core

Web services in WSML-Core follow the common WSML syntax. The logical expressions in the 'assumptions', 'preconditions', 'effects' and 'postconditions' of a capability and 'definiton' of a non-functional property are limited to WSML-Core logical expressions.

3.6. WSML-Core Logical Expression Syntax

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

Let *V* be a WSML-Core vocabulary. Let further γ 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, φ V_I, ψ be either an identifier in V_I or a list of identifiers in V_I, *p,q* V_{RA}, *s,t* 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)*
- φ[*s* **hasValue** *Val*] is an atomic formula in *L(V)*

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

In the following, *F* stands for an lhs-formula (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,...F_n are atomic formulae, then F_1 **and** ... **and** F_n is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_1[*p* hasValue Var_3] **and** Var_3[*p* hasValue Var_2] (globally transitive attribute/relation) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_2[*p* hasValue Var_1] (globally symmetric attribute/relation) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_1[*q* hasValue Var_2] (globally sub-attribute/relation)) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_2[*q* hasValue Var_1] (globally inverse attribute/relation)) is a formula in *L(V)*.
- *G* **equivalent** *H* is a formula in *L(V)* if it contains only one WSML variable.
- *H* **impliedBy** *F* (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* **hasValue** Var_2] in a WSML-Core clause can be interchanged with *p*(Var_1,Var_2) (i.e., these two forms can be used interchangably in WSML Core).

Here, the *variable graph* of a logical expression E is defined as the undirected graph having all WSML variables in E as nodes and an edge between Var_1 and Var_2 for every molecule Var_1[*p* **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 above, 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)*
- 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)*
- φ[*s* **hasValue** *Val*] is an atomic formula in *L(V)*
- If *α* and *β* V_I then *α* = *β* is an atomic formula in *L(V)*

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

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, ..., 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, ..., 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* **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 *β* V_I then **neg** *α* = *β* 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* **hasValue** Var_2] **impliedBy** Var_1[*p* **hasValue** Var_3] **and** Var_3[*p* hasValue Var_2] (globally transitive attribute/relation) is a formula in *L(V)*.

- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_2[*p* **hasValue** Var_1] (globally symmetric attribute/relation) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_1[*q* **hasValue** Var_2] (globally sub-attribute/relation) is a formula in *L(V)*.
- Var_1[*p* **hasValue** Var_2] **impliedBy** Var_2[*q* **hasValue** Var_1] (globally inverse attribute/relation) is a formula in *L(V)*.

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

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

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

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

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

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

- We define the set of admissible head formulae $Head(V)$ as follows:
  - Any atomic formula α which does not contain the inequality symbol (!=) or the unification operator (=) is in $Head(V)$.
  - Let α,β $Head(V)$, 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 β $Head(V)$ and α $Head(V)$ or α $Body(V)$
    - α **impliedBy** β, if α $Head(V)$ and β $Head(V)$ or β $Body(V)$
    - α **equivalent** β if α $Head(V)$ or α $Body(V)$ and β $Head(V)$ or β $Body(V)$
- 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 α,β $Body(V)$, α **and** β is in $Body(V)$.
  - For α,β $Body(V)$, α **or** β is in $Body(V)$.
- Given a head-formula β $Head(V)$ and a body-formula α $Body(V)$, β **:-** α is a formula. Here we call α the *body* and β the *head* of the formula. The formula is admissible if (1) α is an admissible body formula, (2) β is an admissible head formula, and (3) the safety condition holds.
- Any formula of the form !- α with α $Body(V)$ 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 **and** ... **and** A_n **:-** *B* are split into *n* different rules:
  - A_1 **:-** *B*
  - ...
  - A_n **:-** *B*
- Rules of the form A_1 **equivalent** A_2 **:-** *B* are split into 2 rules:
  - A_1 **implies** A_2 **:-** *B*
  - A_1 **impliedBy** A_2 **:-** *B*
- Rules of the form A_1 **impliedBy** A_2 **:-** *B* are transformed to:
  - A_1 **:-** A_2 **and** *B*
- Rules of the form A_1 **implies** A_2 **:-** *B* are transformed to:
  - A_2 **:-** A_1 **and** *B*
- Rules of the form *A* **:-** B_1 **and** (*F* **or** *G*) **and** B_2 are split into two different rules:
  - *A* **:-** B_1 **and** *F* **and** B_2
  - *A* **:-** B_1 **and** *G* **and** B_2
- Rules of the form *A* **:-** B_1 **and naf** (*F* **and** *G*) **and** B_2 are split into two different rules:
  - *A* **:-** B_1 **and naf** *F* **and** B_2
  - *A* **:-** B_1 **and naf** *G* **and** B_2
- Rules of the form *A* **:-** B_1 **and naf** (*F* **or** *G*) **and** B_2 are transformed to:
  - *A* **:-** B_1 **and naf** *F* **and naf** *G* **and** B_2
- Rules of the form *A* **:-** B_1 **and naf naf** *F* **and** B_2 are transformed to:
  - *A* **:-** B_1 **and** *F* **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 hasValue ?x] :- ?x > 25.
?x[gender hasValue male] :- naf ?x[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[gender hasValue {?y, ?z}] memberOf Human and ?y = Male and ?z = Female.
```

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

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

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

- We define the set of admissible head formulae *Head(V)* as follows:
  - Any atomic formula α which does not contain the inequality symbol (!=) or the unification operator (=) is in *Head(V)*.
  - Let *α*,*β* *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 *β* *Head(V)* and *α* *Head(V)* or *α* *Body(V)*
    - *α* **impliedBy** *β*, if *α* *Head(V)* and *β* *Head(V)* or *β* *Body(V)*
    - *α* **equivalent** *β* if *α* *Head(V)* or *α* *Body(V)* and *β* *Head(V)* or *β* *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 *α* *Body(V)*, **naf** *α* is in *Body(V)*.
  - For *α*,*β* *Body(V)*, *α* **and** *β* is in *Body(V)*.
  - For *α*,*β* *Body(V)*, *α* **or** *β* is in *Body(V)*.
  - For *α*,*β* *Body(V)*, *α* **implies** *β* is in *Body(V)*.
  - For *α*,*β* *Body(V)*, *α* **impliedBy** *β* is in *Body(V)*.
  - For *α*,*β* *Body(V)*, *α* **equivalent** *β* is in *Body(V)*.
  - For variables $?x_1,...,?x_n$ and *α* *Body(V)*, **forall** $?x_1,...,?x_n$ (*α*) is in *Body(V)*.
  - For variables $?x_1,...,?x_n$ and *α* *Body(V)*, **exists** $?x_1,...,?x_n$ (*α*) is in *Body(V)*.
- Given a head-formula *β* *Head(V)* and a body-formula *α* *Body(V)*, *β* **:-** *α* is a formula. Here we call *α* the *body* and *β* the *head* of the formula. The formula is admissible if (1) *α* is an admissible body formula, (2) *β* is an admissible head formula.
- Any formula of the form !- *α* with *α* *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.

The following are examples of WSML-Rule logical expressions:

Both the father and the mother are parents:

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

The goal of WSML-Full is to allow the full syntactic freedom of a First-Order logic and the full syntactic freedom of a Logic Programming language with default negation in a common semantic framework. The challenge for WSML-Full is to find an extension of First-Order Logic which can properly capture default negation. One (obvious) possible extension is Reiter's default logic [Reiter, 1987]. However, at the moment the semantics of WSML-Full is still an open research issue. Note that non-monotonic extensions of First-Order Logic are notoriously hard to deal with. In fact, many (if not all) such extensions are not even semi-decidable (i.e., even if the formula is consistent, the algorithm still might not terminate).

Note that both the conceptual and logical expression syntax for WSML-Full completely corresponds with the WSML syntax introduced in Chapter 2. Note also that the WSML-Full logical expression syntax is similar to the logical language specified in WSMO D2 v1.2 [Roman et al., 2005].

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.

PART III: THE WSML EXCHANGE SYNTAXES

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

The three exchange syntaxes for WSML are:

XML syntax:
: A syntax specifically tailored for machine processability, instead of human-readability; it is easily parsable by standard XML parsers, but is quite unreadable for humans. A complete XML syntax for WSML is defined in [Toma, 2008].

RDF syntax
: An alternate exchange syntax for WSML is WSML/RDF. WSML/RDF 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 will be fully defined in [de Bruijn, 2006].

Mapping to OWL
: The mapping to OWL is discussed in [de Bruijn, 2007] and in [Steinmetz, 2008].

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

```
all = [ 0x0 .. 0xffff ]
escape_char = '\'
basechar = [ 0x0041 .. 0x005A ] | [ 0x0061 .. 0x007A ]
ideographic = [ 0x4E00 .. 0x9FA5 ] | 0x3007 | [ 0x3021 .. 0x3029 ]
letter = basechar | ideographic
digit = [ 0x0030 .. 0x0039 ]
combiningchar = [ 0x0300 .. 0x0345 ] | [ 0x0360 .. 0x0361 ] | [ 0x0483 .. 0x0486 ]
extender = 0x00B7 | 0x02D0 | 0x02D1 | 0x0387 | 0x0640 | 0x0E46 | 0x0EC6 | 0x3005 | [ 0x3031 .. 0x3035 ] | [ 0x309D ..
    0x309E ] | [ 0x30FC .. 0x30FE ]
alphanum = digit | letter
hexdigit = [ '0' .. '9' ] | [ 'A' .. 'F' ]
not_escaped_namechar = letter | digit | '_' | combiningchar | extender
escaped_namechar = '.' | '-' | not_escaped_namechar
namechar = ( escape_char escaped_namechar ) | not_escaped_namechar
reserved = '/' | '?' | '#' | '[' | ']' | ';' | ':' | '@' | '&' | '=' | '+' | '$' | ','
mark = '-' | '_' | '.' | '!' | '~' | '*' | ''' | '(' | ')'
escaped = '%' hexdigit hexdigit
unreserved = letter | digit | mark
scheme = letter ( letter | digit | '+' | '-' | '.' )*
port = digit*
idomainlabel = alphanum ( ( alphanum | '-' )* alphanum )?
dec_octet = digit | ( [ 0x31 .. 0x39 ] digit ) | ( '1' digit digit ) | ( '2' [ 0x30 .. 0x34 ] digit ) | ( '25' [ 0x30 .. 0x35 ] )
ipv4address = dec_octet '.' dec_octet '.' dec_octet '.' dec_octet
h4 = hexdigit hexdigit? hexdigit? hexdigit?
ls32 = ( h4 ':' h4 ) | ipv4address
ipv6address = ( ( h4 ':' )* h4 )? '::' ( h4 ':' )* ls32 | ( ( h4 ':' )* h4 )? '::' h4 | ( ( h4 ':' )* h4 )? '::'
ipv6reference = '[' ipv6address ']'
ucschar = [ 0xA0 .. 0xD7FF ] | [ 0xF900 .. 0xFDCF ] | [ 0xFDF0 .. 0xFFEF ]
iunreserved = unreserved | ucschar
ipchar = iunreserved | escaped | ';' | ':' | '@' | '&' | '=' | '+' | '$' | ','
isegment = ipchar*
ipath_segments = isegment ( '/' isegment )*
iuserinfo = ( iunreserved | escaped | ';' | ':' | '&' | '=' | '+' | '$' | ',' )*
iqualified = ( '.' idomainlabel )* '.' ?
ihostname = idomainlabel iqualified
ihost = ( ipv6reference | ipv4address | ihostname )?
iauthority = ( iuserinfo '@' )? ihost ( ':' port )?
iabs_path = '/' ipath_segments
inet_path = '//' iauthority ( iabs_path )?
irel_path = ipath_segments
ihier_part = inet_path | iabs_path | irel_path
iprivate = [ 0xE000 .. 0xF8FF ]
iquery = ( ipchar | iprivate | '/' | '?' )*
ifragment = ( ipchar | '/' | '?' )*
iri_f = scheme ':' ihier_part ( '?' iquery )? ( '#' ifragment )?
tab = 9
cr = 13
lf = 10
eol = cr lf | cr | lf
squote = '''
dquote = '"'
not_cr_lf = [ all - [ cr+ lf ] ]
escaped_char = escape_char all
not_escape_char_not_dquote = [ all - [ '"' + escape_char ] ]
string_content = escaped_char | not_escape_char_not_dquote
long_comment_content = [ all - '/' ] | [ all - '*' ] '/'
long_comment = '/*' long_comment_content* '*/'
begin_comment = '//' | 'comment '
short_comment = begin_comment not_cr_lf* eol
comment = short_comment | long_comment
blank = ( ' ' | tab | eol )+
qmark = '?'
luridel = '_"'
ruridel = '"'
```

Tokens

dollar = '$'
at = '@'
arrow = '=>'
comma = ','
t_pipe = '|'
endpoint = '.' blank
lpar = '('
rpar = ')'
lbracket = '['
rbracket = ']'
lbrace = '{'
rbrace = '}'
hash = '#'
t_and = 'and'
t_or = 'or'
t_implies = 'implies' | '->'
t_implied_by = 'impliedBy' | '<-'
t_equivalent = 'equivalent' | '<->'
t_implied_by_lp = ':-'
t_constraint = '!-'
t_not = 'neg' | 'naf'
t_exists = 'exists'
t_forall = 'forall'
t_univfalse = 'false'
t_univtrue = 'true'
gt = '>'
lt = '<'
gte = '>='
lte = '=<'
equal = '='
unequal = '!='
add_op = '+'
sub_op = '-'
star = '*'
div_op = '/'
t_annotations = 'annotations'
t_assumption = 'assumption'
t_axiom = 'axiom'
t_capability = 'capability'
t_innercapability = 'capability'
t_choreography = 'choreography'
t_statesignature = 'stateSignature'
t_static = 'static'
t_in = 'in'
t_out = 'out'
t_shared = 'shared'
t_controlled = 'controlled'
t_withgrounding = 'withGrounding'
t_transitionrules = 'transitionRules'
t_if = 'if'
t_then = 'then'
t_endif = 'endIf'
t_with = 'with'
t_endforall = 'endForall'
t_choose = 'choose'
t_endchoose = 'endChoose'
t_do = 'do'
t_add = 'add'
t_delete = 'delete'
t_update = 'update'
t_concept = 'concept'
t_definedby = 'definedBy'
t_effect = 'effect'
t_endannotations = 'endAnnotations'
t_ggmediator = 'ggMediator'
t_goal = 'goal'
t_hasvalue = 'hasValue'
t_impliestype = 'impliesType'
t_importontology = 'importsOntology'
t_instance = 'instance'
t_interface = 'interface'
t_innerinterface = 'interface'
t_inverseof = 'inverseOf'
t_subattributeof = 'subAttributeOf'
t_memberof = 'memberOf'
t_namespace = 'namespace'
t_nfp = 'nonFunctionalProperty' | 'nfp'
t_oftype = 'ofType'
t_ontology = 'ontology'
t_oomediator = 'ooMediator'
t_orchestration = 'orchestration'
t_postcondition = 'postcondition'
t_precondition = 'precondition'
t_reflexive = 'reflexive'
t_relation = 'relation'
t_relation_instance = 'relationInstance'

t_sharedvariable = 'sharedVariables'
t_source = 'source'
t_subconcept = 'subConceptOf'
t_subrelation = 'subRelationOf'
t_symmetric = 'symmetric'
t_target = 'target'
t_transitive = 'transitive'
t_usemediator = 'usesMediator'
t_useservice = 'usesService'
t_webservice = 'webService'
t_wgmediator = 'wgMediator'
t_wsmlvariant = 'wsmlVariant'
t_wwmediator = 'wwMediator'
variable = qmark alphanum+
anonymous = '_#'
nb_anonymous = '_#' digit+
pos_integer = digit+
pos_decimal = digit+ '.' digit+
string = dquote string_content* dquote
full_iri = luridel iri_f ruridel
name = (letter | '_') namechar*

Ignored Tokens

- t_blank
- t_comment

Productions

```
wsml = wsmlvariant? namespace? definition*
wsmlvariant = t_wsmlvariant full_iri
namespace = t_namespace prefixdefinitionlist
prefixdefinitionlist = {defaultns} full_iri
    | {prefixdefinitionlist} lbrace prefixdefinition moreprefixdefinitions* rbrace
prefixdefinition = {namespacedef} name full_iri
    | {default} full_iri
moreprefixdefinitions = comma prefixdefinition
definition = {goal} goal
    | {ontology} ontology
    | {webservice} webservice
    | {mediator} mediator
    | {capability} capability
    | {interface} interface
header = {annotations} annotations
    | {usesmediator} usesmediator
    | {importsontology} importsontology
usesmediator = t_usemediator irilist
importsontology = t_importontology irilist
annotations = t_annotations attributevalue* t_endannotations
mediator = {oomediator} oomediator
    | {ggmediator} ggmediator
    | {wgmediator} wgmediator
    | {wwmediator} wwmediator
oomediator = t_oomediator iri? annotations? importsontology? nfp? sources? target? use_service?
ggmediator = t_ggmediator iri? header* nfp? sources? target? use_service?
wgmediator = t_wgmediator iri? header* nfp? source? target? use_service?
wwmediator = t_wwmediator iri? header* nfp? source? target? use_service?
use_service = t_useservice iri?
source = t_source iri?
msources = t_source lbrace iri? moreiris* rbrace
sources = {single} source
    | {multiple} msources
target = t_target iri?
goal = t_goal iri? header* nfp? innercapability? innerinterfaces*
webservice = t_webservice id? header* nfp? innercapability? innerinterfaces*
capability = t_capability iri? header* nfp? sharedvardef? pre_post_ass_or_eff*
innercapability = t_innercapability iri? header* nfp? sharedvardef? pre_post_ass_or_eff*
sharedvardef = t_sharedvariable variablelist
pre_post_ass_or_eff = {precondition} t_precondition axiomdefinition
    | {postcondition} t_postcondition axiomdefinition
    | {assumption} t_assumption axiomdefinition
    | {effect} t_effect axiomdefinition
interfaces = {single} interface
    | {multiple} minterfaces
innerinterfaces = {single} innerinterface
    | {multiple} minnerinterfaces
minterfaces = t_interface lbrace id moreiris* rbrace
minnerinterfaces = t_innerinterface lbrace id moreiris* rbrace
interface = t_interface id? header* nfp? choreography? orchestration?
innerinterface = t_innerinterface iri? header* nfp? choreography? orchestration?
choreography = t_choreography iri? header* statesignature?
statesignature = t_statesignature id? header* mode*
mode = mode_id mode_entry_list
mode_entry_list = {mode_entry} mode_entry
    | {mode_entry_list} mode_entry comma mode_entry_list
mode_id = {static} t_static
    | {in} t_in
    | {out} t_out
    | {shared} t_shared
    | {controlled} t_controlled
mode_entry = {default_mode} iri grounding?
    | {concept_mode} t_concept iri grounding?
    | {relation_mode} t_relation iri grounding?
grounding = t_withgrounding grounding_info
grounding_info = {iri} iri
    | {irilist} lbrace listiri rbrace
listiri = {iri} iri
    | {listiri} iri comma listiri
transitions = t_transitionrules id? annotations? rule*
rule = {if} t_if condition t_then rule+ t_endif
    | {forall} t_forall variablelist t_with condition t_do rule+ t_endforall
    | {choose} t_choose variablelist t_with condition t_do rule+ t_endchoose
    | {uncond} piped_rules
    | {undate} updaterule
condition = {restricted_le} expr
```

```
piped_rules =      {rule} rule
              | {piped} rule  t_pipe  piped_rules
updaterule =  modifier  lpar  fact  rpar
modifier =        {add} t_add
              |  {delete} t_delete
              | {update} t_update
fact =        {fact_preferred} term  attr_fact?  t_memberof  termlist  fact_update?
              | {fact_nonpreferred} term  t_memberof  termlist  fact_update?  attr_fact
              |      {fact_molecule} term  attr_fact
              |        {fact_relation} at  id  lpar  term_updates  rpar
fact_update =  arrow  termlist
attr_fact =  lbracket  attr_fact_list  rbracket
attr_fact_list =  {attr_relation} term  t_hasvalue  termlist  fact_update?
              |                  attr_fact_list  comma  term  t_hasvalue  termlist  fact_update?
term_updates =     {one_param} term_update
              | {more_params} term_update  comma  term_updates
term_update =  {single} term
              | {move} [oldterm]:  term  arrow  [newterm]:  term
new_term = {new_term} arrow  term
orchestration =  t_orchestration  iri?
nfp =  t_nfp  attributevaluenfp
attributevaluenfp =  id  t_hasvalue  valuelistnfp  header?  log_definition?
valuelistnfp =         {term} instattvaluenfp
              | {valuelistnfp} lbrace  instattvaluenfp  moreinstattvaluesnfp*  rbrace
instattvaluenfp =  {instattvalue} instattvalue
              |           {var} variable
moreinstattvaluesnfp =  comma  instattvaluenfp
ontology =  t_ontology  iri?  header*  ontology_element*
ontology_element =         {concept} concept
              |         {instance} instance
              |          {relation} relation
              | {relationinstance} relationinstance
              |            {axiom} axiom
concept =  t_concept  id  superconcept?  annotations?  attribute*
superconcept =  t_subconcept  idlist
att_type =    {open_world} t_oftype
              | {closed_world} t_impliestype
attribute =  id  attributefeature*  att_type  cardinality?  idlist  annotations?
cardinality =  {constraining} lpar  pos_integer  cardinality_number?  rpar
              |       {infering} lbracket  pos_integer  cardinality_number?  rbracket
cardinality_number =    {finite_cardinality} pos_integer
              | {infinite_cardinality} star
attributefeature =         {transitive} t_transitive
              |        {symmetric} t_symmetric
              |           {inverse} t_inverseof  lpar  id  rpar
              | {sub_attribute_of} t_subattributeof  lpar  id  rpar
              |          {reflexive} t_reflexive
instance =  t_instance  id?  memberof?  annotations?  attributevalue*
memberof =  t_memberof  idlist
attributevalue =  id  t_hasvalue  valuelist
relation =  t_relation  id  arity?  paramtyping?  superrelation?  annotations?
paramtyping =  lpar  paramtype  moreparamtype*  rpar
paramtype =  att_type  idlist
moreparamtype =  comma  paramtype
superrelation =  t_subrelation  idlist
arity =  div_op  pos_integer
relationinstance =  t_relation_instance  [name]:  id?  [relation]:  id  lpar  value  morevalues*  rpar  annotations?
axiom =  t_axiom  axiomdefinition
axiomdefinition =         {use_axiom} id
              | {annotations_axiom} id?  annotations
              |      {defined_axiom} id?  annotations?  log_definition
log_definition =  t_definedby  log_expr+
log_expr =              {lp_rule} [head]:  expr  t_implied_by_lp  [body]:  expr  endpoint
              |        {constraint} t_constraint  expr  endpoint
              | {other_expression} expr  endpoint
expr =  {implication} expr  imply_op  disjunction
              | {disjunction} disjunction
disjunction =  {conjunction} conjunction
              |              disjunction  t_or  conjunction
conjunction =  {subexpr} subexpr
              |              conjunction  t_and  subexpr
subexpr =    {negated} t_not  subexpr
              |    {simple} simple
              |   {complex} lpar  expr  rpar
              | {quantified} quantified
quantified =  quantifier_key  variablelist  lpar  expr  rpar
simple =     {molecule} molecule
              | {comparison} comparison
```

```
                 | {atom} term
molecule =         {concept_molecule_preferred} term attr_specification? cpt_op termlist
                 | {concept_molecule_nonpreferred} term cpt_op termlist attr_specification
                 | {attribute_molecule} term attr_specification
attr_specification = lbracket attr_rel_list rbracket
attr_rel_list =    {attr_relation} attr_relation
                 | attr_rel_list comma attr_relation
attr_relation =    {attr_def} term attr_def_op termlist
                 | {attr_val} term t_hasvalue termlist
comparison =       [left]: term comp_op [right]: term
functionsymbol =   {parametrized} id lpar terms? rpar
                 | {math} lpar arith_val rpar
arith_val =        mult_val
                 | {addition} arith_val arith_op mult_val
                 | {semisimple1_addition} term arith_op mult_val
                 | {semisimple2_addition} arith_val arith_op term
                 | {simple_addition} [a]: term arith_op [b]: term
mult_val =         [a]: term mul_op [b]: term
                 | {multiplication} mult_val mul_op term
math_op =          {arith} arith_op
                 | {mult} mul_op
arith_op =         {add} add_op
                 | {sub} sub_op
mul_op =           {mul} star
                 | {div} div_op
comp_op =          {gt} gt
                 | {lt} lt
                 | {gte} gte
                 | {lte} lte
                 | {equal} equal
                 | {unequal} unequal
cpt_op =           {memberof} t_memberof
                 | {subconceptof} t_subconcept
quantifier_key =   {forall} t_forall
                 | {exists} t_exists
attr_def_op =      {oftype} t_oftype
                 | {impliestype} t_impliestype
imply_op =         {implies} t_implies
                 | {impliedby} t_implied_by
                 | {equivalent} t_equivalent
prefix =           name hash
compactURI =       {any} prefix? name
                 | {localkeyword} prefix anykeyword
anykeyword =       {and} t_and
                 | {or} t_or
                 | {implies} t_implies
                 | {implied_by} t_implied_by
                 | {equivalent} t_equivalent
                 | {implied_by_lp} t_implied_by_lp
                 | {constraint} t_constraint
                 | {not} t_not
                 | {exists} t_exists
                 | {forall} t_forall
                 | {univfalse} t_univfalse
                 | {univtrue} t_univtrue
                 | {annotations} t_annotations
                 | {assumption} t_assumption
                 | {axiom} t_axiom
                 | {capability} t_capability
                 | {choreography} t_choreography
                 | {concept} t_concept
                 | {definedby} t_definedby
                 | {effect} t_effect
                 | {end_annotations} t_endannotations
                 | {ggmediator} t_ggmediator
                 | {goal} t_goal
                 | {hasvalue} t_hasvalue
                 | {impliestype} t_impliestype
                 | {importontology} t_importontology
                 | {instance} t_instance
                 | {interface} t_interface
                 | {inverseof} t_inverseof
                 | {memberof} t_memberof
                 | {namespace} t_namespace
                 | {nfp} t_nfp
                 | {oftype} t_oftype
                 | {ontology} t_ontology
                 | {oomediator} t_oomediator
                 | {orchestration} t_orchestration
                 | {postcondition} t_postcondition
                 | {precondition} t_precondition
                 | {reflexive} t_reflexive
                 | {relation} t_relation
                 | {relation_instance} t_relation_instance
```

```
            | {sharedvariable} t_sharedvariable
            |         {source} t_source
            |     {subconcept} t_subconcept
            |    {subrelation} t_subrelation
            |      {symmetric} t_symmetric
            |         {target} t_target
            |     {transitive} t_transitive
            |   {usemediator} t_usemediator
            |     {useservice} t_useservice
            |     {webservice} t_webservice
            |     {wgmediator} t_wgmediator
            |    {wsmlvariant} t_wsmlvariant
            |     {wwmediator} t_wwmediator
        iri =              {iri} full_iri
            | {compactURI} compactURI
         id =              {iri} iri
            | {anonymous} anonymous
     irilist =     {iri} iri
            | {irilist} lbrace  iri  moreiris*  rbrace
    moreiris =  comma  iri
      idlist =     {id} id
            | {idlist} lbrace  id  moreids*  rbrace
     moreids =  comma  id
       anyid =  {datavalue} datavalue
            |          {id} id
       value =  {datatype} functionsymbol
            |      {term} id
            |  {numeric} number
            |     {string} string
 instattvalue =              {term} id
            |            {numeric} number
            |             {string} string
            |                {iri} iri  t_hasvalue  valuelist
            |       {anonymous} anonymous  t_hasvalue  valuelist
            | {nb_anonymous} nb_anonymous  t_hasvalue  valuelist
   valuelist =      {term} value
            | {valuelist} lbrace  value  morevalues*  rbrace
  morevalues =  comma  value
        term =              {data} value
            |                {var} variable
            | {nb_anonymous} nb_anonymous
       terms =  {term} term
            |        terms  comma  term
    termlist =  {term} term
            |        lbrace  terms  rbrace
   variables =  {variable} variable
            |            variables  comma  variable
 variablelist =      {variable} variable
            | {variable_list} lbrace  variables  rbrace
     integer =  sub_op?  pos_integer
     decimal =  sub_op?  pos_decimal
      number =   {integer} integer
            | {decimal} decimal
   datavalue =        {simpledatavalue} simpledatavalue
            | {constructeddatavalue} constructeddatavalue
datavaluelist =  {simpledatavalue} simpledatavalue
            |     {datavaluelist} lbrace  simpledatavalue  moresimpledatavalues*  rbrace
moresimpledatavalues =  comma  simpledatavalue
 simpledatavalue =  {number} number
            |    {string} string
constructeddatavalue =  {iri} iri  lbrace  datavaluelist  rbrace
```

A.2. Example of the Human-Readable Syntax

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

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

/******************************
 * ONTOLOGY
 *****************************/
ontology _"http://www.example.org/ontologies/example"
    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"
    wsml#version hasValue "$Revision: 1.15 $"
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
    hasParent inverseOf(hasChild) impliesType Human
    hasChild subAttributeOf(hasRelative) impliesType Human
    hasAncestor transitive impliesType Human
    hasRelative symmetric impliesType Human
    hasWeight ofType (1) xsd#decimal
    hasWeightInKG ofType (1) xsd#decimal
    hasBirthdate ofType (1) xsd#date
    hasObit ofType (0 1) xsd#date
    hasBirthplace ofType (1) loc#location
    isMarriedTo symmetric impliesType (0 1) Human
    hasCitizenship ofType oo#country
    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 */
        hasWeighthasWeightInKG hasValue 3.52
        hasBirthdate hasValue xsd#date(2004,10,21)

    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"
        wsml#version hasValue "$Revision: 1.15 $"
    endAnnotations

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

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

    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 ?brithdate]
                    and wsml#dateLessThan(?birthdate,wsml#currentDate())
                    and ?child[hasBirthplace hasValue ?location]
                    and ?location[locatedIn hasValue oo#de]
                    or (?child[hasParent hasValue ?parent]
                      and?parent[hasCitizenship hasValue oo#de] ) .

        assumption
            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"
        wsml#version hasValue "$Revision: 1.15 $"
    endAnnotations

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

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

    capability
        sharedVariables ?human
        effect havingACitzienShip
            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"
        wsml#version hasValue "$Revision: 1.15 $"
    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"
        wsml#version hasValue "$Revision: 1.15 $"
    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"
        wsml#version hasValue "$Revision: 1.15 $"
```

```
    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].

Table B.1: WSML Datatypes

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
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(*integer_year*, *integer_month*, *integer_day*, *integer_hour*, *integer_minute*, *integer_second*, *integer_timezone-hour*, *integer_timezone-minute*) xsd#dateTime(*integer_year*, *integer_month*, *integer_day*, *integer_hour*, *integer_minute*, *integer_second*)	_dateTime
time		xsd#time(*integer_hour*, *integer_minute*, *integer_second*, *integer_timezone-hour*, *integer_timezone-minute*) xsd#time(*integer_hour*, *integer_minute*, *integer_second*)	_time
date		xsd#date(*integer_year*, *integer_month*, *integer_day*, *integer_timezone-hour*, *integer_timezone-minute*) xsd#date(*integer_year*, *integer_month*, *integer_day*)	_date
gYearMonth		xsd#gYearMonth(*integer_year*, *integer_month*)	_gyearmonth
gYear		xsd#gYear(*integer_year*)	_gyear
gMonthDay		xsd#gMonthDay(*integer_month*, *integer_day*)	_gmonthday
gDay		xsd#gDay(*integer_day*)	_gday
gMonth		xsd#gMonth(*integer_month*)	_gmonth
hexBinary		xsd#hexBinary("*hexadecimal-encoding*")	_hexbinary
base64Binary		xsd#base64Binary("*base64-encoding*")	_base64binary
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]**.	rdf#XMLLiteral("*"-escaped XML content*")	

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

Table B.2: WSML Datatype shortcut syntax

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

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.

Table B.3: WSML Built-in Predicates

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:time-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	
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:

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

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)
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.

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

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)

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

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

1. Select an atomic occurrence of a datatype function symbol. An atomic occurrence is an occurrence of the function symbol with only identifiers (which can be variables) as arguments.
2. Replace this occurrence with a newly introduced variable and append to the conjunction of which the function 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 and wsml#numericAdd(?x1, ?x, ?y)
```

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

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

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

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

Appendix 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.

[TODO: check keyword/variant combo]Table C.1: WSML keywords

Keyword	Section	Core	Flight	Rule	Full
wsmlVariant	2.2.1	+	+	+	+
namespace	2.2.2	+	+	+	+
annotations	2.2.3	+	+	+	+
endAnnotations	2.2.3	+	+	+	+
importsOntology	2.2.3	+	+	+	+
usesMediator	2.2.3	+	+	+	+
ontology	2.3	+	+	+	+
concept	2.3.1	+	+	+	+
subConceptOf	2.3.1	+	+	+	+
ofType	2.3.1.1	+	+	+	+
impliesType	2.3.1.1	+	+	+	+
transitive	2.3.1.1	+	+	+	+
symmetric	2.3.1.1	+	+	+	+
inverseOf	2.3.1.1	+	+	+	+
subAttributeOf	2.3.1.1	+	+	+	+
reflexive	2.3.1.1		+	+	+
relation	2.3.2	+	+	+	+
subRelationOf	2.3.2	+	+	+	+
instance	2.3.3	+	+	+	+
memberOf	2.3.3	+	+	+	+
hasValue	2.3.3	+	+	+	+
relationInstance	2.3.3	+	+	+	+
axiom	2.3.4	+	+	+	+
definedBy	2.3.4	+	+	+	+
capability	2.4.1	+	+	+	+
sharedVariables	2.4.1	+	+	+	+
precondition	2.4.1	+	+	+	+
assumption	2.4.1	+	+	+	+
postcondition	2.4.1	+	+	+	+
effect	2.4.1	+	+	+	+
interface	2.4.2	+	+	+	+
choreography	2.4.2	+	+	+	+

orchestration	2.4.2	+	+	+	+
nonFunctionalProperty	2.4.3	+	+	+	+
nfp	2.4.3	+	+	+	+
goal	2.5	+	+	+	+
ooMediator	2.6	+	+	+	+
ggMediator	2.6	+	+	+	+
wgMediator	2.6	+	+	+	+
wwMediator	2.6	+	+	+	+
source	2.6	+	+	+	+
target	2.6	+	+	+	+
usesService	2.6	+	+	+	+
webService	2.7	+	+	+	+

Table 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

Table C.2: WSML logical expression keywords

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

Appendix D. Changelog

D.1. Issues List

For a list of the issues to be addressed in this version of D16.1, see the issues list.

D.2. Changes in the Language since Version 0.21

- The concept of Nonfunctional properties in version 0.21 has been split into two notions: annotations and nonfunctional properties.
- A syntax for choreographies, based on WSMO deliverable 14 is now included in the language.
- The datatypes _iri and _sqname are removed in. IRIs are abstract symbols interpreted arbitrarily, in contrast to data values which are concrete symbols interpreted according to a particular datatype. Depicting IRI as a datatype is misleading in this respect.
- Datatype wrappers are now IRIs; the names of datatypes constructors used in the previous version (e.g. `_date`) may be used as shortcuts.
- It is no longer allowed to use an anonymous identifier for the identification of the following elements: ontology, goal, Web service, mediator, capability, interface, import ontologies, used mediators, sources and targets of mediators, services used by mediators, choreography, orchestration
- local identifiers have been introduced (and have been removed again, see D.3.
- New datatype rdf#XMLLiteral is introduction
- The symbol :=: (strong equality) has been removed from WSML 0.3
- Equality and negated equality (a=b, neg a=b) has been allowed in WSML-DL logical expressions for instances.
- Relations have been removed from WSML-Core and WSML-DL
- sQNames have been renamed to Compact URIs
- Non functional properties are not written as blocks anymore, but one by one, using the nonfunctionalproperty or nfp keywords. Furthermore importsOntology and usesMediator are disallowed for nfps now.
- Imported ontologies can be WSML ontologies, RDF graphs or OWL DL and OWL Full ontologies.

Changes in the Language implemented in version 0.3 published on 17th of March 2008

- The authors mentioned in the list are now the WSML working group members.
- A new, shorter, abstract is included.
- The text in appendix C.1 is updated to reflect the updates in the treatment of datatypes.
- The section on identifiers in Chapter 2 is updated to reflect the changes in the treatment of datatypes.
- The text about local stratification, which was included there by mistake, in the WSML Rule chapter is removed.
- Appendix E. "Relation to WSMO Conceptual Model" is removed
- Syntax for choreographies from D14 was added in 2.4.2
- Chapter 10 "Implementation Efforts" is removed
- Chapter 8 "XML Syntax for WSML" and chapter 9 "RDF Syntax for WSML" are removed; the XML syntax for wsml will be discussed in a new wsml deliverables; the RDF syntax for wsml will be discussed in D32
- Syntax for NFPs was changed to allow both values and variables followed by logical expressions
- Section 4.6.1, Alternative Surface Syntax for WSML-DL, has been removed.
- All references to new dl-based conceptual syntax, as e.g. concept Thing, keyword completely definedBy, etc., have been removed.
- The section on identifiers in Chapter 2 includes also variables.
- A notion has beend added to WSML-DL to make clear that WSML-DL does not adhere to the Unique Name Assumption.
- Added explanations to imported ontologies section, to clarify that not only WSML ontologies but also RDFS and OWL ontologies can be imported.

D.3. Changes in This Document since the Previous Version

- Removed link to XML syntax at the top of the document
- Added explanations to imported ontologies section, to clarify that not only WSML ontologies but also RDFS and OWL ontologies can be imported.
- Updated introduction to WSML Language reference in explaining the relationship between WSMO, WSML Abstract Syntax and WSML Language Reference, i.e. this document. Furthermore added references to the xml, rdf and owl exchange syntaxes, respectively mappings.
- Minor fixes concerning the use of non functional properties and the use of optional identifiers.

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.

[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, 2006] J. de Bruijn (Ed). *RDF representation of WSML*. WSML Working Draft D32v0.1, 2006. Available from http://www.wsmo.org/TR/d32/v0.1/.

[de Bruijn, 2007] J. de Bruijn (Ed). *WSML Abstract Syntax*. WSML Working Draft D16.3v0.3, 2007. 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.

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

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

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

[Keller et al., 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., 2005] 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 Discover*y, 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 ASG, DIP, enIRaF, InfraWebs, Knowledge Web, Musing, Salero, SEKT, Seemp, SemanticGOV, Super, SWING and TripCom; by Science Foundation Ireland under the DERI-Lion Grant No.SFI/02/CE1/I13 ; by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects Grisino, RW^2, SemNetMan, SeNSE and TSC.

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

Footnotes

[1]The work presented in [Levy & Rousset, 1998] might serve as a starting point to define a subset of WSML-Full which could be used to enable a higher degree of interoperation between Description Logics and Logic Programming (while retaining decidability, but possibly losing tractability) rather than through their common core described in WSML-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.