D16.1v0.2 The WSML Family of Representation Languages

WSML Working Draft 26 November 2004

This version
: http://www.wsmo.org/2004/d16/d16.1/v0.2/20041126/

Latest version
: http://www.wsmo.org/2004/d16/d16.1/v0.2/

Previous version
: http://www.wsmo.org/2004/d16/d16.1/v0.2/20041025/

Editor:
Jos de Bruijn

Authors:
Jos de Bruijn
Holger Lausen
Dieter Fensel

Reviewers:
Ian Horrocks
Jeff Pan

Copyright © 2004 DERI ®, All Rights Reserved. DERI liability, trademark, document use, and software licensing rules apply.

// Abstract

We introduce WSML, a family of formal representation languages with its roots in Description Logics, First-Order Logic and Logic Programming. The conceptual modeling elements of WSML are based on the meta-model of WSMO.

WSML itself consists of a number of variants which are based on the different underlying logical formalisms.

The variant **WSML-Core** semantically corresponds with the intersection of Description Logic and Horn Logic, extended with extensive datatype support in order to be useful in practical applications. WSML-Core is fully compliant with a subset of OWL, albeit that the datatype support in WSML-Core is already beyond OWL, because the datatype support in OWL is very limited.

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

WSML-DL extends WSML-Core to an expressive Description Logic, namely *SHOIN*.

WSML-Flight extends WSML-Core in the direction of Logic Programming with more intuitive value restrictions and cardinality constraints. WSML-Flight is the preferred ontology modeling language, because of its rich set of modeling primitives for modeling different aspects of attributes, such as value constraints and integrity constraints, and its rich logical language which allows for writing down arbitrary rules. Furthermore, WSML-Flight incorporates a fully-fledged rule language, while still allowing efficient decidable reasoning.

WSML-Rule extends WSML-Flight to a fully-fledged Logic Programming language, including function symbols and higher-order features of HiLog and possible Transaction Logic.

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

WSML-Full unifies all WSML variants under a common First-Order umbrella with non-monotonic extensions.

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

This deliverable supersedes a number of now obsolete WSML deliverables. References to these now obsolete deliverables can be found at: http://www.wsmo.org/2004/d16/

Table of Contents

PART I: PRELUDE

1. Introduction

- 1.1. Structure of the deliverable

PART II: WSML VARIANTS

2 Common WSML Syntax

- 2.1 WSML Syntax Basics
  - 2.1.1 Namespaces in WSML
  - 2.1.2 Identifiers in WSML
  - 2.1.3 Datatypes in WSML
- 2.2 Common WSML Elements
  - 2.2.1 WSML Variant declaration
  - 2.2.2 Namespace References
  - 2.2.3 Non-functional properties
  - 2.2.4 Importing Ontologies
  - 2.2.5 Using Mediators
- 2.3 Ontology Specification in WSML
- 2.4 Goal Specification in WSML
  - 2.4.1 Goal Postconditions
  - 2.4.2 Goal Effects
- 2.5 Mediator Specification in WSML
  - 2.5.1 ooMediators
  - 2.5.2 wwMediators
  - 2.5.3 ggMediators
  - 2.5.4 wgMediators
- 2.6 Web Service Specification in WSML
  - 2.6.1 Web Service Capability
  - 2.6.2 Web Service Interface

3 WSML-Core

- 3.1 Basic WSML-Core Syntax
- 3.2. WSML-Core Elements
  - 3.2.1 Concepts
  - 3.2.2 Relations
  - 3.2.3 Functions
  - 3.2.4 Instances
  - 3.2.5 Axioms
  - 3.2.6 User-defined Datatypes
- 3.3. Goals in WSML-Core
- 3.4. Mediators in WSML-Core
- 3.5. Web Services in WSML-Core
- 3.6. WSML-Core Logical Expressions
- 3.7. Conclusions

4 WSML-Flight

- 4.1 Basic WSML-Flight Syntax
- 4.2. WSML-Flight Ontologies

    - 4.2.1 Attributes
    - 4.2.2 Relations
    - 4.2.3 Functions
    - 4.2.4 Relation Instances
- 4.3. WSML-Flight Logical Expression Syntax
- 4.4. Goals in WSML-Flight
- 4.5. Mediators in WSML-Flight
- 4.6. Web Services in WSML-Flight
- 4.7. Differences between WSML-Core and WSML-Flight

5 WSML-Rule

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

6 WSML-DL

- 6.1 Basic WSML-DL Syntax
- 6.2. WSML-DL Ontologies
- 6.3. WSML-DL Logical Expression Syntax
- 6.4. Goals in WSML-DL
- 6.5. Mediators in WSML-DL
- 6.6. Web Services in WSML-DL
- 6.7. Differences between WSML-Core and WSML-DL

7 WSML-Full

- 7.1 Basic WSML-Full Syntax
- 7.2. WSML-Full Ontologies
- 7.3. WSML-Full Logical Expression Syntax
- 7.4. Goals in WSML-Full
- 7.5. Mediators in WSML-Full
- 7.6. Web Services in WSML-Full
- 7.7. Differences between WSML-DL and WSML-Full
- 7.8. Differences between WSML-Rule and WSML-Full

PART III: THE WSML EXCHANGE SYNTAXES

8 XML Syntax for WSML

- 8.1. Future Work

9 RDF Syntax for WSML

- 9.1 Representing Ontologies using RDF
- 9.2 Representing WSML using RDF

10 Mapping to OWL

- 10.1. Mapping WSML-Core to OWL DL-
- 10.2. Mapping OWL DL- to WSML-Core

PART IV: FINALE

11. Related Efforts

12. Conclusions

References

Acknowledgements

Appendix A. Grammar for the human-readable syntax

- A.1. BNF-style grammar for WSML
- A.2. An example of WSML in the human-readable syntax

Appendix B. XML Schemas for the XML exchange syntax

- B.1. XML Schema for the XML syntax of WSML
- B.2. An example of a WSML/XML ontology

Appendix C. RDF Schemas for the RDF exchange syntax

Appendix D. Built-ins in WSML

- D.1. WSML Datatype predicates
- D.2. Translating built-in symbols to Datatype predicates

Appendix E. A list of all WSML keywords

Appendix F. Changelog and Discussion Issues

- F.1. Changelog
- F.2. Discussion Issues

PART I: PRELUDE

1. Introduction

Ontologies and Semantic Web Services need formal languages for their specification in order to enable automatic processing. The W3C recommendation for an ontology language OWL [Dean & Schreiber, 2004] has serious limitations both on a conceptual level and with respect to some of its formal properties [de Bruijn et al., 2004]. 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., 2004]. For example, OWL-S offers the choice between different language for the specification of preconditions and effect. However, it is not entirely clear how these languages interact with OWL, which is used for the specification of inputs and output.

The Web Service Modeling Ontology WSMO [Roman *et al.* 2004] proposes a conceptual model for the description of Ontologies and Semantic Web Services and Ontologies, providing the conceptual grounding for Ontology and Web Service description. In this deliverable we take the conceptual model of WSMO as a starting point for the specification of a family of Web Service description and Ontology specification languages, by the name of WSML. The different variants of WSML correspond with different levels of logical expressiveness and the use of different styles of logical languages. More specifically, we take Description Logics, First-Order Logic and Logic Programming as starting points for the development of the various WSML variants. We provide a clean layering of the WSML variants, both syntactically and semantically. All WSML variants are specified in terms of a human-readable syntax with keywords similar to the elements of the WSMO conceptual model. Furthermore, we provide an XML syntax as an exchange language, as well as an RDF syntax and a mapping to OWL for interoperability with RDF- and OWL-based applications.

1.1. Structure of the Deliverable

This deliverable is structured as follows.

PART II: WSML VARIANTS

Chapter 2 describes the common elements between the WSML variants, as well as syntax basics, such as the use of namespaces and the basic vocabulary of the languages. 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 describes WSML-Flight, which is an extension of WSML-Core in the direction of Logic Programming. WSML-Flight is a more powerful logic language and offers more powerful modeling constructs than WSML-Core. Chapter 5 describes WSML-Rule which is a full-blown Logic Programming language, but it is still an extension of WSML-Flight and thus it offers the same kind of conceptual modeling features. Chapter 6 presents WSML-DL, which is an extension of WSML-Core. The extension described by WSML-DL is disjoint from the extensions described by WSML-Flight and WSML-Rule. WSML-DL is a full-blown Description Logic and offers the same kind of expressive power as OWL DL [Patel-Schneider et al., 2004]. Chapter 7, finally, presents WSML-Full which is a superset of both WSML-Rule and WSML-DL. WSML-Full can be seen as a notational variant of First-Order Logic with nonmonotonic extensions.

PART III: THE WSML EXCHANGE SYNTAXES

The WSML variants are described in terms of their normative human-readable language in PART II. Although this syntax has been formally specified in the form of a grammar in Appendix A, there are limitations with respect to the exchange of the syntax between machines. Therefore, Chapter 8 presents the XML exchange syntax for WSML, which is the preferred syntax for the exchange of WSML specifications between machines. Chapter 9 describes the RDF syntax of WSML in order to allow for basic interoperation with RDF-based applications. Chapter 10, finally, described a partial mapping to OWL in order to allow interoperation with OWL-based applications.

PART IV: FINALE

We conclude the deliverable with describing efforts related to the WSML languages in Chapter 11. These related efforts are mostly concerned with implementation of WSML-based tools and tools utilizing WSML for specific purposes. Chapter 12, finally, presents conclusions and future work.

Appendix Guide

This deliverable contains a large number of appendices. It is therefore worthwhile to describe here the content of these 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 to this deliverable. Appendix B contains references to the XML Schemas, the XML Schema documentation and the XML version of the WSML example of Appendix A. These documents are all online resources. Future versions of this deliverable might integrate (parts of) these resources. However, these resources take up a lot of space and blow up the size of this deliverable. Appendix C will present the RDF Schema for the RDF syntax of WSML. Appendix D describes the built-in predicates of WSML, as well as a set of infix operators which correspond with built-in predicates. Appendix E contains a complete list of WSML keywords, as well as references to the places in this deliverable where they are described. Finally, Appendix F contains the changelog.

PART II: WSML VARIANTS

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

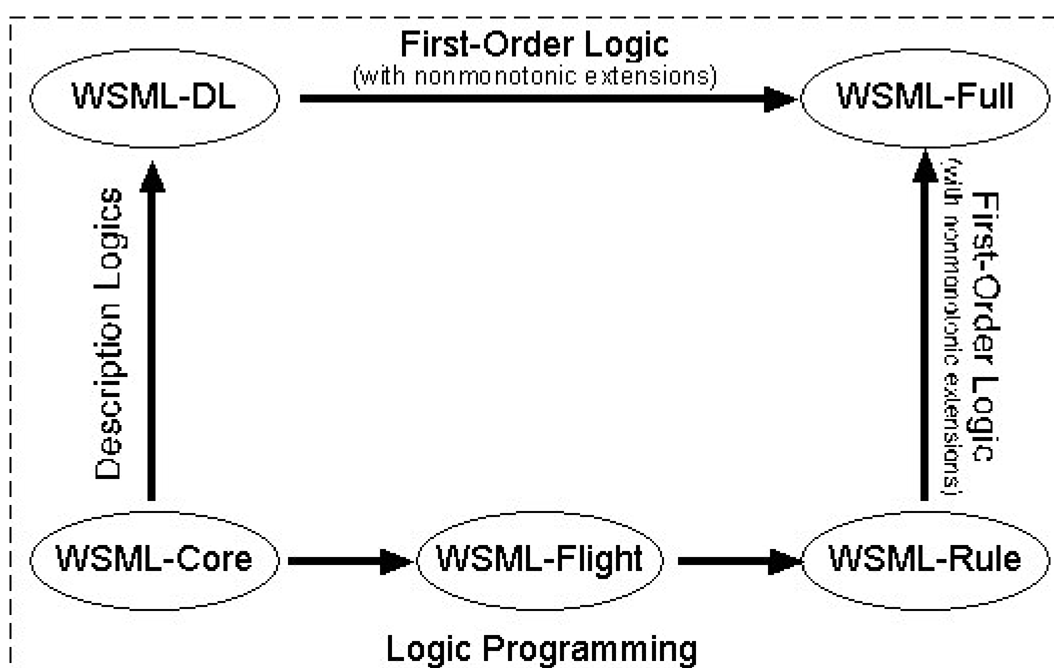

Figure 1. WSML Space

WSML-Core

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

WSML-Flight

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

WSML-Rule

This language will be an extension of WSML-Flight in the direction of Logic Programming. The language will capture several extensions such as the use of

function symbols and possibly extensions based on HiLog [Chen et al., 1993] and Transaction Logic [Bonner & Kifer, 1998], which are required for rich Web Service discovery [Keller et al., 2004]. Another possible extension for WSML-Rule is disjunction in the head of the rule, as in Disjunctive Datalog [Eiter et al., 1997].

WSML-DL

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

WSML-Full

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

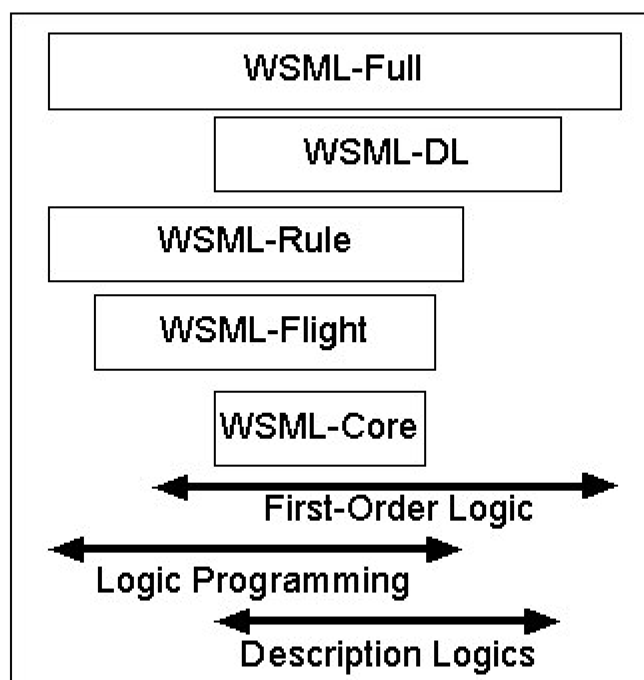

Figure 2. WSML Layering

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

The only languages currently specified in this document are WSML-Core and WSML-Flight. WSML-Rule is currently underspecified, because the requirements on the language are not yet clear. WSML-DL will correspond (semantically) with the Description Logic *SHOIN*(D_n), extended with more extensive datatype support. WSML-Full will provide the formal semantics for the logical language specified in WSMO D2 [Roman et

al., 2004]. Notice that D2 only provides an intuitive semantics for the logical language.

In the descriptions in the subsequent chapters we use fragments of the WSML grammar (A.1) in order to show the syntax of the WSML elements. The grammar is specified using a dialect of Extended BNF which can be used direction in the SableCC compiler compiler [SableCC]. Terminals are quoted, non-terminals are underlined and refer to the corresponding productions. Alternatives are separated using vertical bars '|', and may be 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 '+'.

2 Common WSML Basics

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

This chapter is structured as follows. We first introduce basics of the WSML syntax, such as the use of namespaces, identifiers, etc. in Section 2.1. Then we describe the elements all WSML variants have in common in Section 2.2. We describe the modeling of ontologies, goals, mediators and web services in section 2.3, 2.4, 2.5 and 2.6, respectively.

2.1 WSML Syntax basics

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

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

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

A WSML specification is separated in two parts. The first part is the meta-information about the specification, which consists of such things as WSML variant identification, namespace references, non-functional properties (annotations), the type of the specification, import of ontologies and references to used mediators. This header of the specification is strictly ordered. The body of the specification, consisting of elements such as concepts, attributes, relations (in the case of an ontology specification), capability, interfaces (in the case of a web service specification), etc., is not ordered.

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

2.1.1 Namespaces in WSML

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

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

Notice that namespaces only provide a syntactical separation of names.

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

The target namespace of an ontology is the namespace in which the elements in the specification are defined by default. It is good practice to use as the target namespace the identifier of the ontology and to define all elements in the specification in this namespace.

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

2.1.2 Identifiers in WSML

Identifiers in WSML is either an IRI [Duerst & Suignard, 2004], a literal, a variable or an anonymous id.

Internationalized Resource Identifiers

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

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

full_iri = ‘<"’ iri_reference ‘">’

qname = (ncname ':')**?** ncname

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

Literals

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

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

WSML allows the following syntactical shortcuts for literals:

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

```
literal      =   {typedliteral} typedliteral
               | {plainliteral} plainliteral
               | {numeric}      number
               | {string}       string
plainliteral = '"'  literal_content*  '"'  ( language_tag )?
typedliteral =   plainliteral  '^^'  id
```

Variables

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

The scope of a variable always corresponds with its quantification. If a variable is not quantified inside a formula, the variable is free.

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

Anonymous identifiers

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

Numbered anonymous IDs are denoted with '_#*n*' where *n* stands for an integer denoting the number of the anonymous ID. All occurrences of a particular numbered anonymous

ID in the same *scope* refer to the same object. Each occurrence of an unnumbered anonymous ID can be seen as a new unique identifier. Each numbered anonymous ID can be seen as a new unique identifier which shared inside its scope.

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

Certain occurrences of unnumbered anonymous IDs ('_#') can be disregarded, namely when the unnumbered anonymous ID is an identifier of:

- A **relationInstance**, since an instance of a relation simply consists of a tuple, identified by its parameter values.
- The parameter name for arguments of a **relationInstance** in case the relation does not have named parameters.

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

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

- In case the identifier of an ontology, web service, goal or mediator is omitted, the identifier is assumed to be the same as the locator of the specification, i.e. the location where the specification can be found. Notice that in case no explicit target namespace has been specified, that this is also the target namespace of the specification.
- In case the identifier of a WSML element (e.g. concept, relation, postcondition) has been omitted, the anonymous identifier '_#' is used to identify the element.

```
id =   {iri}                  full_iri
     | {qname}                qname
     | {literal}              literal
     | {anonymous}            anonymous
     | {var}                  variable
     | {universal_truth}      'true'
     | {universal_falsehood}  'false'
```

In case the same identifier is used for different definitions, it is interpreted differently, depending on the context. In a concept definition, an identifier is interpreted as a concept; in a relation definition this same identifier is interpreted as a relation. If, however, the same identifier is used in separate definitions, but with the same context, then the interpretation of the identifier has to conform to both definitions and thus the definitions are interpreted conjunctively. For example, if there are two concept definitions which are concerned with the same concept identifier, the resulting concept definition includes all attributes of the original definitions and if the same attribute is defined in both definitions, the range of the resulting attribute will be equivalent to the conjunction of the original attributes.

Definition 2.1. A WSML vocabulary V consists of:

- A set of identifiers V_{ID}
- A set of abstract identifiers V_{AID} which is a subset of V_{ID}. V_{AID} consists of all IRI references, QNames and anonymous identifiers.
- A set of IRI references V_{IRI} which is a subset of V_{AID}. V_{IRI} consists of all full IRIs references and QNames.
- A set of concept identifiers V_C which is a subset of V_{ID}. V_C contains *wsml:true* and *wsml:false*.
- A set of datatype identifiers V_D which is a subset of V_{ID}. V_D consists of the XML Schema basic datatypes, *rdfs:Literal* and all user-defined datatypes.
- A set of identifiers for relations V_R which is a subset of V_{ID}
- A set of identifiers for relations with unnamed parameters V_{RU} which is a subset of V_R
- A set of identifiers for relations with named parameters V_{RN} which is a subset of V_R
- A set of identifiers for functions with unnamed parameters V_{FU} which is a subset of V_{RU}
- A set of identifiers for functions with named parameters V_{FN} which is a subset of V_{RN}
- A set of parameter identifiers V_{Par} which is a subset of V_{IRI}
- A set of attribute identifiers V_{Att} which is a subset of V_{RU}
- A set of instance identifiers V_I which is a subset of V_{ID}
- A set of literals V_L which is a subset of V_{ID}
- A set of axiom identifiers V_A which is a subset of V_{AID}
- A set of non-functional property identifiers V_{NFP} which is a subset of V_{ID}
- A set of ontology identifiers V_O which is a subset of V_{AID}
- A set of goal identifiers V_G which is a subset of V_{AID}
- A set of web service identifiers V_{WS} which is a subset of V_{AID}
- A set of capability identifiers V_{Cap} which is a subset of V_{AID}
- A set of interface identifiers V_{In} which is a subset of V_{AID}
- A set of choreography identifiers V_{Cho} which is a subset of V_{AID}
- A set of orchestration identifiers V_{Orc} which is a subset of V_{AID}
- A set of mediator identifiers V_M which is a subset of V_{AID}
- A set of ooMediator identifiers V_{OOM} which is a subset of V_M
- A set of ggMediator identifiers V_{GGM} which is a subset of V_M
- A set of wgMediator identifiers V_{WGM} which is a subset of V_M
- A set of wwMediator identifiers V_{WWM} which is a subset of V_M

V_{RU} and V_{RN} are disjoint.

2.1.3 Datatypes in WSML

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

In order to use a datatype, the datatype must be *known*, which means that datatype identifiers are known to refer to a datatype. All XML Schema built-in datatypes [Biron & Malhorta, 2004] are *known* datatypes. In case the user wants to use a datatype other than an XML Schema built-in datatype, the datatype has to be made known using the **datatype** construct (see Section 3.2.6).

2.2 Common WSML Elements

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

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

A WSML document consists of the following:

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

2.2.1 WSML Variant

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

WSML Variant	IRI
WSML-Core	http://www.wsmo.org/2004/wsml/wsml-core
WSML-Flight	http://www.wsmo.org/2004/wsml/wsml-flight
WSML-Rule	http://www.wsmo.org/2004/wsml/wsml-rule
WSML-DL	http://www.wsmo.org/2004/wsml/wsml-dl
WSML-Full	http://www.wsmo.org/2004/wsml/wsml-full

Table 2.1: WSML variant identifiers

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

```
wsmlvariant = 'wsmlVariant' full_iri
```

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

```
wsmlVariant <"http://www.wsmo.org/2004/wsml/wsml-core">
```

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

2.2.2 Namespace References

At the top of a WSML document, below the identification of the WSML variant, there is an optional block of namespace references, which is preceded by the **namespace** keyword. The **namespace** keyword a number of namespace references. Each namespace reference, except for the default namespace, consists of the chosen prefix, a colon ':' and the IRI which identifies the namespace. Finally, the (optional) target namespace is specified with the use of the **targetNamespace** keyword. If the target namespace coincides with the identifier of the ontology (which is the usual case), the **targetNamespace** line is redundant and can be omitted. Notice that, like any argument list in WSML, the list of namespace references is delimited with curly brackets '{' '}'.

```
namespace             = 'namespace'  prefixdefinitionlist
prefixdefinitionlist  = {defaultns}            full_iri
                        | {prefixdefinitionlist} '{'  prefixdefinition  moreprefixdefinitions*  '}'
prefixdefinition      = {namespacedef} ncname  full_iri
                        | {default}            full_iri
moreprefixdefinitions = ','  prefixdefinition
targetnamespace       = 'targetNamespace'  full_iri
```

An example:

```
namespace {<"http://www.example.org/example/">,
  dc: <"http://purl.org/dc/elements/1.1#">,
  xsd: <"http://www.w3.org/2001/XMLSchema#"> }

targetNamespace: <"http://www.example.org/example/">
```

2.2.3 Non-functional properties

Non-functional properties may be used for the WSML document as a whole but also for each element in the specification. Non-functional property blocks are identified by the keyword **nonFunctionalProperties** or **nfp**. Following the keyword is a list of attribute values, which consists of the attribute identifier, the keyword **hasValue** and the value for the attribute, which may be any identifier and can thus be an IRI, a literal or an anonymous identifier. The recommended properties are the properties of the Dublin Core [Weibel et al. 1998], but the list of properties is extensible and thus the user can choose to use properties coming from different sources. WSMO [Roman et al., 2004] defines a number of properties which are not in the Dublin Core. These properties can be used in a WSML specification by referring to the WSML namespace (http://www.wsmo.org/2004/wsml). These properties are: **wsml:version**, **wsml:accuracy**, **wsml:financial**, **wsml:networkRelatedQoS**, **wsml:performance**, **wsml:reliability**, **wsml:robustness**, **wsml:scalability**, **wsml:security**, **wsml:transactional**, **wsml:trust** (here we assume that the prefix wsml has been defined as referring to the WSML namespace). For recommended usage of these properties see [Roman et al., 2004]. Following the

WSML convention, if a property has multiple values, these are separated by commas and the list of values is delimited by curly brackets.

```
header = {nfp}                  nfp
       | {usedmediators}        usedmediators
       | {importedontologies}   importedontologies
nfp    = 'nfp' attributevalue* 'endnfp'
       | 'nonFunctionalProperties' attributevalue* 'endNonFunctionalProperties'
```

An example:

```
nonFunctionalProperties
  dc:title hasValue "WSML example collection"
  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"> }
  dc:date hasValue "2004-09-20"
  dc:type hasValue <"http://www.wsmo.org/2004/d2/v1.0/#ontologies">
  dc:format hasValue "text/plain"
  dc:language hasValue "en-US"
  dc:rights hasValue <"http://www.deri.org/privacy.html">
  version hasValue "$Revision: 1.18 $"
endNonFunctionalProperties
```

2.2.4 Importing Ontologies

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

```
importedontologies = 'importedOntologies' idlist
```

An example:

```
importedOntologies
   {<"http://www.example.org/ontologies/dateTime#">,
   <"http://www.example.org/ontologies/currency#">}
```

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

2.2.5 Using Mediators

Mediators are used to link different WSML elements (ontologies, goal, web services) and resolve heterogeneity between the elements. Mediators are described in more detail in Section 2.5. We are here only concerned with how mediators can be referenced from a WSML specification.

The (optional) used mediators block is identified by the keywords **usedMediators** which is followed by one or more identifiers of WSMO mediators. The types of mediators that can be used are constrained by the type of specification. An ontology allows for the use of different mediators than, for example, a goal or a web service. 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.

usedmediators = 'usedMediators' idlist

An example:

```
usedMediators
  { <"http://www.example.org/mediators/dateTime"> ,
    <"http://www.example.org/mediators/trainConnection"> ,
    <"http://www.example.org/mediators/purchase"> ,
    <"http://www.example.org/mediators/location"> }
```

2.3 Ontology Specification in WSML

A WSML ontology specification is identified by the **ontology** keyword optionally followed by a IRI which serves as the identifier of the ontology. When no explicit target namespace definition exists, this identifier is used as the target namespace of the definitions in the ontology specification document. When no identifier is specified for the ontology, the locator of the ontology serves as identifier.

An example:

```
ontology <"http://wsmo.org/ontologies/purchase/">
```

An ontology specification document in WSML consists of:

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

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

2.4 Goal Specification in WSML

A WSML goal specification is identified by the **goal** keyword optionally followed by a IRI which serves as the identifier of the goal. When no explicit target namespace definition exists, this identifier is used as the target namespace of the definitions in the goal specification document. When no identifier is specified for the goal, the locator of the

goal serves as identifier.

An example:

```
goal <"http://www.wsmo.org/2004/d3/d3.3/v0.1/20041008/resources/goal1.wsml">
```

A goal specification document in WSML consists of:

```
goal                  = 'goal' id? nfp? usedmediators? importedontologies*
                        postcondition_or_effect*
postcondition_or_effect = {postcondition} 'postCondition' axiomdefinition
                       | {effect}         'effect'   axiomdefinition
```

The elements of a goal are postconditions and effects which are explained below.

2.4.1 Goal Postconditions

A postcodition starts with the **postcondition** keyword, (optionally) followed by the name (identifier) of the postcondition. This is followed by an optional **nonFunctionalProperties** block and then by a logical expression preceded by the **definedBy** keyword. The language allowed for the logical expression differs per WSML variant and is explained in the respective chapters.

An example of a postcondition:

```
postcondition havingItineraryForTrip
 nonFunctionalProperties
   dc:description hasValue "This goal expresses the general desire of buying a ticket for
     any kind of trip. Thus, the goal postcondition defines that there shall be an
     itinerary for a trip for a passenger that is a person."
 endNonFunctionalProperties
 definedBy
   ?someItinerary memberOf tc:itinerary[
     tc:trip hasValue ?someTrip,
     tc:passenger hasValue ?somePassenger]
   and ?someTrip memberOf tc:trip
   and ?somePassenger memberOf loc:person.
```

2.4.2 Goal Effects

An effect starts with the **effect** keyword, (optionally) followed by the name (identifier) of the effect. This is followed by an optional **nonFunctionalProperties** block and then by a logical expression preceded by the **definedBy** keyword. The language allowed for the logical expression differs per WSML variant and is explained in the respective chapters.

An example of an effect specified in WSML-Full:

```
effect havingTradeForTrip
  nonFunctionalProperties
    dc:description hasValue "The goal effect is to have a trade with a provider (not specified)
      for the itinerary given; the buyer and the payment method are specified according
      to the Purchase ontology."
  endNonFunctionalProperties
  definedBy
    ?sometrade memberOf po:trade[
      items hasValue {?someTrip},
      buyer hasValue ?someBuyer,
      payment hasValue ?somePayment
```

```
]
and ?someBuyer memberOf po:buyer
and ?somePayment memberOf po:paymentMethod.
```

Notice that in goals, logical expressions only occur in postconditions and effects.

Besides the logical expressions in postconditions and effects, none of the elements of a goal definition have a meaning in a logical languages. WSML also does not prescribe a relationship between postconditions and effects. It is up to the user of the definitions to use them appropriately. WSML deliverable D5.1 [Keller et al., 2004] contains suggestions on how to use these elements for Web Service discovery.

Ontology imported by the goal, either through an **importedOntologies** or a **usedMediators** statement, are logically appended to both the postcondition and the effect. The resulting postcondition is the union of the axiom on the postcondition and the set of axioms in the ontolog(y)(ies). Similar for the effect. Notice that a mediator used to import an ontology typically contains axioms which resolve heterogeneity between ontologies. From the point-of-view of the goal, these logical axioms are part of the imported ontology and thus also of the logical union with the postcondition/effect.

2.5 Mediator Specification in WSML

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

A WSML mediator specification is identified by the one of the keywords identifying a particular kind of mediator (**ooMediator**, **wwMediator**, **ggMediator**, **wgMediator**), optionally followed by a IRI which serves as the identifier of the mediator. When no explicit target namespace definition exists, this identifier is used as the target namespace of the definitions in the mediator specification document. When no identifier is specified for the mediator, the locator of the mediator serves as identifier.

An example:

```
ooMediator <"http://www.wsmo.org/2004/d3/d3.3/v0.1/20041008/resources/owlAddressMediator.wsml">
```

A mediator specification document in WSML consists of:

```
mediator =   {oomediator} oomediator
           | {ggmediator} ggmediator
           | {wgmediator} wgmediator
           | {wwmediator} wwmediator
```

2.5.1 ooMediators

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.

oomediator = 'ooMediator' id? nfp? importedontologies? source* target? 'mediationService'?

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

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

2.5.2 wwMediators

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

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

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

2.5.3 ggMediators

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

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

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

2.5.4 wgMediators

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

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

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

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

None of the elements in a mediator has any meaning in the logical language. In fact, the complexity of a mediator is hidden in the actual description of the mediator. Instead, the complexity is either in the implementation of the mediation service, in which case WSML does not support the description because WSML is only concerned with the interface description, or in the functionality description of the mediation service or the goal which is used to specify the desired mediation service. As is discussed in [Keller et al., 2004], these descriptions often need a very expressive language. Furthermore, for the ontology mapping task, which is typically required for ontology mediation, an expressive language is often required [de Bruijn & Polleres, 2004].

2.6 Web Service Specification in WSML

A WSML web service specification is identified by the **webService** keyword optionally followed by a IRI which serves as the identifier of the web service. When no explicit target namespace definition exists, this identifier is used as the target namespace of the definitions in the web service specification document. When no identifier is specified for the web service, the locator of the web service specification serves as identifier.

An example:

webService <"http://www.wsmo.org/2004/d3/d3.3/v0.1/20041008/resources/ws.wsml">

A web service specification document in WSML consists of:

webservice	=	'webService' id? nfp? usedmediators? importedontologies* capability? interface*

The elements of a web service are capability and interface which are explained below.

2.6.1 Web Service Capability

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

A capability description starts with the **capability** keyword, (optionally) followed by the name (identifier) of the capability. This is followed by an optional **nonFunctionalProperties** block, an optional **importOntologies** block and an optional **usedMediators** block and then by a number of **assumption**, **preCondition**, **effect**, and **postCondition** definitions. The number of such definitions is not restricted. Each of these definitions consists of the keyword, an optional identifier, an optional **nonFunctionalProperties** block and a logical expression preceded by the **definedBy** keyword. The language allowed for the logical expression differs per WSML variant and is explained in the respective chapters.

capability	=	{use_capability}	'useCapability' id
		| {defined_capability}	capabilitydef
capabilitydef	=	'capability' id? nfp? usedmediators? importedontologies? pre_post_ass_or_eff*	
pre_post_ass_or_eff	=	{precondition}	'preCondition' axiomdefinition
		| {assumption}	'assumption'> axiomdefinition
		| {post_or_effect}	postcondition_or_effect

An example of a capability specified in WSML-Full (for reasons of space we only show one assumption; a capability typically also has preconditions, postconditions and

effects):

```
capability
  precondition
    nonFunctionalProperties
      dc:description hasValue "the input has to be a buyer with a purchase intention for
        an itinerary for which the start- and endlocation have to be in Austria
        or in Germany, and the departure date has to be later than the current Date.
        A credit card as payment method is expected."
    endNonFunctionalProperties
    definedBy
      ?Buyer memberOf po:buyer and
      ?Trip memberOf tc:trainTrip[
        tc:start hasValue ?Start,
        tc:end hasValue ?End,
        tc:departure hasValue ?Departure
      ] and
      (?Start.locatedIn = austria or ?Start.locatedIn = germany) and
      (?End.locatedIn = austria or ?End.locatedIn = germany) and
      dt:after(?Departure,currentDate).
```

2.6.2 Web Service Interface

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

An interface specification starts with the **interface** keyword, (optionally) followed by the name (identifier) of the interface. This is followed by an optional **nonFunctionalProperties** block, an optional **importOntologies** block and an optional **usedMediators** block and then by an optional choreography block consisting of the keyword **choreography** followed by the identifier of the choreography and an optional orchestration block consisting of the keyword **orchestration** followed by the identifier of the orchestration. Notice that thus an interface can have at most one choreography and at most one orchestration.

```
interface     =   {use_interface}     'useInterface'  id
                | {defined_interface} interfacedef
interfacedef  =   'interface'  id?  nfp?  usedmediators?  importedontologies?
                  choreography?  orchestration?
choreography  =   'choreography'  id
orchestration =   'orchestration'  idlist
```

An example of an interface:

```
interface
  nonFunctionalProperties
    dc:description hasValue "describes the Interface of Web Service (not specified yet)"
  endNonFunctionalProperties
  choreography _"http://www.example.org/mychoreography"
  orchestration _"http://www.example.org/myorchestration"
```

Besides logical expressions in the capability, none of the elements of a web service definition have a meaning in a logical language, although we expect elements of WSML ontologies and/or logical expressions to also be used in choreographies and orchestrations. WSML also does not prescribe a relationship between these elements of the capability. It is up to the user of the definitions to use them appropriately. WSML

deliverable D5.1 [Keller et al., 2004] contains suggestions on how to use these elements for Web Service discovery.

3 WSML-Core

The major goal of the WSML working group is to develop a formal language for the description of Semantic Web Services based on the WSMO conceptual model. As described in the introduction to this Part, there are several WSML languages with different underlying logical formalisms. The two main logical formalisms exploited in different WSML languages 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 semantics of OWL DL- [de Bruijn et al., 2004], which is based on the Logic Programming subset of Description Logics described in [Grosof et al., 2003]. Furthermore, WSML-Core uses a restricted form of the OWL-E datatype extension [Pan and Horrocks, 2004] of OWL. The relation between this extension and OWL DL- is described in [de Bruijn et al., 2004a]. The modeling constructs of WSML-Core are based on the conceptual model of WSMO [Roman et al., 2004]. Because WSML-Core is based on OWL DL- and there exists a translation from OWL DL- to plain (function- and negation-free) Datalog, the decidability and complexity results of Datalog apply to WSML-Core as well. The most important result is that Datalog is data complete for P, which means that query answering can be done in polynomial time.[2]

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

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

This chapter is structured as follows. We first introduce basics of the WSML-Core syntax, such as the use of namespaces, identifiers, etc. in Section 3.1. We describe 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 logical expression syntax of WSML-Core in Section 3.6.

3.1 Basic WSML-Core Syntax

WSML-Core inherits the basic of the WSML syntax specified in Section 2.1. In this section we describe restrictions WSML-Core poses on the basic syntax.

WSML-Core inherits the namespace mechanism of WSML.

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

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

Definition 3.1. A WSML-Core vocabulary *V* is a WSML vocabulary according to Definition 2.1. with the following additional restrictions:

- V_C, V_D, V_R, V_I and V_{NFP} are subsets of V_{AID}.
- V_{Par} = { *wsml:domain*, *wsml:domain* }
- $V_{Att} = V_{RU}$
- A set of relations with an abstract range V_{RA} which is a subset of V_R.
- A set of relations with a concrete range V_{RC} which is a subset of V_R.
- $V_R = V_{RA} \cup V_{RC}$
- V_{RA} and V_{RC} are disjoint
- V_C, V_D, V_R, V_I and V_{NFP} are pairwise disjoint

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

3.2. WSML-Core Ontologies

In this section we explain the ontology modeling elements in the WSML-Core language. The modeling elements are based on the WSMO conceptual model of ontologies [Roman et al., 2004]. Each description is accompanied by an example. Some of the examples were based on [Stollberg et al., 2004].

3.2.1 Concepts

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

concept	=	'concept' id superconcept? nfp? attribute* log_definition?
superconcept	=	'subConceptOf' idlist

An example:

```
concept Human subConceptOf {Primate, LegalAgent}
   nonFunctionalProperties
      dc:description hasValue "Members of the species Homo sapiens"
   endNonFunctionalProperties
   parent impliesType Human
   length ofType xsd:integer
   weight ofType xsd:integer
   bmi ofType xsd:float
```

In the example, the attributes length and weight have as their type the datatype xsd:integer. bmi has datatype xsd:float. BMI, in this case, stands for Body Mass Index, which is a certain ratio between the length and the the weight of a person.

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

subconcept and attribute definitions.

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

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

?x **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:

```
concept Human
   definedBy
      ?x memberOf Human equivalent ?x memberOf Primate and ?x memberOf LegalAgent.
```

3.2.1.1 Attributes

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

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

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

Concept attributes (i.e. attributes that do not have a datatype as range) can be specified as being transitive, symmetric, or being the inverse of another attribute, using the **transitive**, **symmetric** and **inverseOf** keywords, respectively. Notice that these keywords do not enforce a constraint on the attribute, but are used to infer additional information about the attribute. The keyword **inverseOf** must be followed by an identifier

of the attribute, enclosed in parentheses, of which this attribute is the inverse.

```
att_type            =  {constraint} 'ofType'
                     | {restriction} 'impliesType'
attribute           =  [attr]: id att_type attributefeature* [type]: id nfp?
cardinality_number  =  {finite_cardinality} pos_int
                     | {infinite_cardinality} '*'
attributefeature    =  {transitive} 'transitive'
                     | {symmetric} 'symmetric'
                     | {inverse} 'inverseOf' '(' id ')'
```

[TODO: rephrase paragraph] 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*.

3.2.2 Relations

A relation definition starts with the **relation** keyword, which is optionally followed by the identifier of the relation. This is optionally followed by a superrelation definition which consists of the keyword **subRelationOf** followed by one or more relation identifiers (as usual, if there is more than one, the list is comma-separated and delimited by curly brackets). This is followed by an optional **nonFunctionalProperties** block. In WSML-Core, relations are restricted to binary predicates, which correspond with ObjectProperties and DatatypeProperties in OWL. Parameters can be used to restrict the domain and range of the property (denoted by the **domain** and **range** keywords, respectively), i.e. from the domain and range restrictions, class membership is inferred. Besides **domain** and **range**, no other parameters are allowed.

```
relation       =  'relation' id superrelation? nfp? parameter* log_definition?
superrelation  =  {relation} 'subRelationOf' idlist
parameter      =  [parameter]: id att_type [type]: id
```

The (optional) parameters of the relation, namely **domain** and **range**, are followed by the keyword **impliesType** to reflect the fact that concept membership is implied from such specifications.

An example:

```
relation hasAncestor subRelationOf hasRelative
    nonFunctionalProperties
        dc:description hasValue "Relation between ancestors"
    endNonFunctionalProperties
    domain impliesType Person
    range impliesType Person
```

Besides defining a relation over two concepts, it is also possible to define a relation over

a concept and a datatype, corresponding to the DatatypeProperties in OWL. Such relations over a concept and a datatype do not use a different keyword, but may only be a subrelation of another relation over a concept and a datatype. In case the range of a relation is a datatype, the keyword **ofType** must be used to reflect that the range is restricted to values of the specific datatype and the parameter does not allow for deriving membership of a certain datatype.

An example:

```
relation length subRelationOf hasMeasure
    nonFunctionalProperties
        dc:description hasValue "Length indicator"
    endNonFunctionalProperties
    range ofType xsd:integer
```

3.2.3 Functions

WSML-Core allows the user to create functions, which correspond with relations with an n-ary domain and a unary range. In WSML-Core, both domain and range may only be datatypes. In WSML-Core, both the domain and the range must be datatypes. A function definitions starts with the **function** keyword, followed by the name (identifier) of the function. This is followed by an optional **nonFunctionalProperties** block and then by an optional range and an optional set of parameters and a datatype expression preceded by the **definedBy** keyword. The syntax for the datatype expressions is explained in Section 3.

function	=	'function' id superrelation? nfp? functionparameters? log_definition?
functionparameters	=	parameter* 'range' att_type id

The example below defines a new predicate for calculating the Body Mass Index. The symbols used for the datatype functions are '/', '=' and '*'. These symbols stand for numerical division, equality and multiplication, respectively. Notice that these symbols are shortcuts for built-in datatype predicates. The translation of symbols to datatype predicates is given in Appendix D.1.

An example of a function:

```
function bodyMassIndex
    nonFunctionalProperties
        dc:description hasValue "Calculates the Body Mass Index.
    endNonFunctionalProperties
    length ofType xsd:integer
    weight ofType xsd:integer
    range ofType xsd:float
    definedBy
        bodyMassIndex(range hasValue ?bmi, length hasValue ?length, weight hasValue ?weight) impliedBy
          ?bmi = ?weight / ?sqLength and ?sqLength = ?length * ?length.
```

The careful reader may have noticed that the function definition in the example can typically not be evaluated by a reasoner implementation because the datatypes xsd:integer and xsd:float are infinite. Thus, computation of the extension of the function bodyMassIndex would take infinite time. In order to resolve this situation, the definition of the function is substituted for every occurrence of the function (i.e. every time this function is called).

3.2.4 Instances

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

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

An example:

```
instance innsbruckHbf memberOf station
    name hasValue 'Innsbruck Hauptbahnhof'
    code hasValue "INN"^^xsd:string
    locatedIn hasValue loc:innsbruck
```

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

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

An example:

```
relationInstance memberOf hasAge
    domain hasValue John
    range hasValue 23
```

relationinstance	=	'relationInstance' [relation]: id 'memberOf' [type]: id nfp? attributevalue*

3.2.5 Axioms

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

axiom = 'axiom' axiomdefinition
axiomdefinition = {use_axiom} id
| {defined_axiom} id? nfp? log_definition
log_definition = 'definedBy' log_expr

An example of a defining axiom:

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

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

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

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

3.2.6 User-defined Datatypes

WSML-Core allows the user to create user-defined datatypes. A datatype is completely defined through a logical expression of the form:

datatype expression **and** ... **and** *datatype expression*.

Each *datatype expression* is either a datatype membership expression or a datatype predicate. Notice that built-in predicates ranging over data values, such as '<' and '>=', are considered to be datatype predicates. The syntax for datatype expressions is detailed further in Section .... In case a datatype has several definitions, these definitions are interpreted conjunctively. Similar as for functions, the datatype expression is substituted for every reference to the datatype.

datatype = 'datatype' axiomdefinition

An example of a user-defined datatype:

```
datatype positiveInt
    nonFunctionalProperties
        dc:description hasValue "Positive integers."
    endNonFunctionalProperties
    definedBy
        ?x memberOf positiveInt impliedBy ?x memberOf xsd:integer and ?x > 0.
```

3.3. Goals in WSML-Core

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

3.4. Mediators in WSML-Core

Goals in WSML-Core follow the common WSML syntax.

3.5. Web Services in WSML-Core

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

3.6. WSML-Core Logical Expression Syntax

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

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

WSML has the following simple logical expressions:

- Relation Expressions
- Molecules

WSML has the following complex logical expressions:

- Compound Logical Expressions
- Formulas

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

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

3.6.1 Simple Logical Expressions

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

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

3.6.1.1 Relation Expressions

A relation logical expression consists of a predicate identifier followed by the comma-separated arguments of the predicate, enclosed by parentheses. A relation value logical expression is one that can be expressed by a single binary relation relating the two arguments. Both arguments can either be data values (i.e. literals) or identifiers referring to instances. However, if the second argument is a data value, the first argument must also be a data value. In the latter case, the relation actually corresponds to a datatype predicate. More about datatype predicates in Section 2.3.3.

An example:

ageInYears(john, 23)

Relations in WSML-Core may be parameterized with the parameters **domain** and **range**. If the relation is not parameterized, the first argument is interpreted as the domain and the second argument as the range. The advantage of using the parameters **domain** and **range** is that order of the arguments is no longer important.

An example which is equivalent to the previous example:

ageInYears(**range hasValue** 23, **domain hasValue** john)

3.6.1.2 Molecules

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

An instance molecule is one of the following statements:

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

A concept membership assertion of the form *I* **memberOf** *C* is true iff *I* is an instance of concept *C*. An attribute value list specifies the values for certain attributes for this particular instance.

Two examples:

myCC **memberOf** creditCard

myCC[number **hasValue** '12345', owner **hasValue** jos]

A concept molecule is one of the following statements:

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

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

attribute *A* is of the type *D*.

Two examples:

creditCard **subConceptOf** paymentMethod

creditCard[number **ofType** xsd:string, owner **ofType** person]

3.6.2 Complex Logical Expressions

WSML-Core has the following complex logical expressions:

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

3.6.2.1 Compound Logical Expressions

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

An example:

myCC **memberOf** creditCard **and** myCC[number **hasValue** '12345', owner **hasValue** jos]

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

myCC[number **hasValue** '12345', owner **hasValue** jos] **memberOf** creditCard

3.6.2.2 Formulas

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

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

wrt. the same variable binding. A dual implication E_1 **<->** E_3 is actually equivalent to the two implications: E_1 **<-** E_3 and E_1 **->** E_3. Therefore, both E_1 and E_3 are both the *antecedent* and the *consequent* of the formula.

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

An example:

?x **memberOf** Human **<->** ?x **memberOf** Animal **and** ?x **memberOf** LegalAgent.

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

false impliedBy ?x **memberOf** Human **and** ?x[age **hasValue** ?y] **and not** ?y **memberOf** xsd:string.

This integrity constraint is violated if there is any human with an age which is not of type xsd:string.

3.6.3 Datatype Expressions

3.6.3.1 Datatype predicates

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

An example:

wsml:numeric-equals(3,4)

The user is free to choose the built-in predicates as long as the implementation knows how to handle the built-ins (similar to the allowed datatypes). However, we recommend a minimal list of supported datatype predicates, which are listed in Appendix D.1. These predicates operate on the domains of xsd:string and xsd:integer, which are the basic datatypes, which should be supported by any WSML-Core implementation.

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

3.6.3.2 Compound Datatype Expressions

TODO: compound datatype expressions need to be explained and their relation with regular logical expressions needs to be investigated Keywords are: **and**, **or**, **memberOf** and **not**.

3.7. Conclusions

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

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

4. WSML-Flight

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

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

The features added by WSML-Flight are the following:

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

4.1 Basic WSML-Flight Syntax

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

4.2. WSML-Flight Ontologies

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

4.2.1 Attributes

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

Besides the more general use of the **ofType** keyword, WSML-Flight adds two distinct features to attribute definitions compared with WSML-Core. The first feature is reflexivity. Reflexivity is asserted by using the **reflexive** keyword. The second feature is the cardinality constraints. The cardinality constraints for a single attribute are specified by including two numbers between parentheses '(' ')', indicating the minimal and maximal cardinality, after the **ofType** keyword. The first number indicates the minimal cardinality.

The second number indicates the maximal cardinality, where '*' stands for unlimited maximal cardinality (and is not allowed for minimal cardinality). It is possible to write down just one number instead of two, which is interpreted as both a minimal and a maximal cardinality constraint. When the cardinality is omitted, then it is assumed that there are no constraints on the cardinality, which is equivalent to (0 *). Notice that a maximal cardinality of 1 makes an attribute functional.

attribute	=	[attr]: id att_type cardinality? attributefeature* [type]: id nfp?
cardinality	=	'(' pos_int cardinality_number? ')'
cardinality_number	=	{finite_cardinality} pos_int | {infinite_cardinality} '*'
attributefeature	=	{transitive} 'transitive' | {symmetric} 'symmetric' | {inverse} 'inverseOf' '(' id ')' {reflexive} 'reflexive'

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

```
concept Human
    subConceptOf { Primate, LegalAgent }
    nonFunctionalProperties
       dc:description hasValue "Members of the species Homo sapiens"
    endNonFunctionalProperties

    ancestor ofType (0 *) transitive
    parentOf ofType inverseOf(hasParent) Human
    hasParent ofType (1) inverseOf(parentOf) Human
    authorOf ofType inverseOf(hasAuthor) Publication
    isRelated ofType symmetric Human
    length ofType (0 1) xsd:integer
    weight ofType (0 1) xsd:integer
    bmi ofType (0 1) xsd:float
```

4.2.2 Relations

Relations in WSML-Flight can have an arbitrary arity. Furthermore, the parameters of a relation in WSML-Flight can have arbitrary names. The parameters of a relation in WSML-Flight can be of a constraining fashion (using **ofType**) besides the inferred fashion (using **impliesType**) inherited from WSML-Core. Finally, there are no restrictions on the use of datatypes as types for parameters, although it is still not possible to specify an inferring parameter (**impliesType**) for a datatype.

An example:

```
relation distance subRelationOf measurement
    from ofType Location
    to ofType Location
    distance ofType xsd:integer
```

4.2.3 Functions

Functions in WSML-Flight can now also be specified for non-datatype parameters. In case such a function does contain datatypes in its domain, it is not necessary to

substitute the definition of the function for each occurrence of the function, because variable are bound by non-built-in predicates.

An example:

```
function daysBetween
  nonFunctionalProperties
    dc:description hasValue "(Instant1, Instant2, Difference) is a triple of the ternary relation
      corresponding to this function iff Instant1 and Instant2 are members of the concept
      instant (particular point in time) and Instant2 is Difference days after Instant1."
  endNonFunctionalProperties
  instant1 ofType timeInstant
  instant2 ofType timeInstant
  range ofType xsd:integer
  definedBy
    daysBetween[instant1 hasValue ?D1, instant2 hasValue ?D2, result hasValue ?X] impliedBy
      ?D1 memberOf date and ?D2 memberOf date and
      ?X = julianDayNumber(?D1) - julianDayNumber(?D2).
```

4.2.4 Relation Instances

Because WSML-Flight allows relations of arbitrary arity, as well as arbitrary parameters for relations, also the relation instances in WSML-Flight are slightly different from WSML-Core. In case the relation does not have any parameters, the name of the parameter in the relation instance definition is the anonymous identifier '_#'. The arguments of a relation without parameters are ordered. The order of parameter values in the relation instance definition corresponds with the order of arguments in the relation definition.

An example of an instance of a ternary relation with no parameters (remember that the identifier is optional, see also Section 2.1.2):

```
relationInstance memberOf distance
   _# hasValue innsbruckHbf
   _# hasValue munichHbf
   _# hasValue 234
```

4.3. WSML-Flight Logical Expression Syntax

TODO: to be defined; is based on Datalog with stratified negation

4.4. Goals in WSML-Flight

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

4.5. Mediators in WSML-Flight

Mediators in WSML-Core follow the common WSML syntax.

4.6. Web Services in WSML-Flight

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

4.7. Differences between WSML-Core and WSML-Flight

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

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

5. WSML-Rule

WSML-Rule is not yet specified; it is not yet clear what the specific requirements are. Most likely, WSML-Rule will have unrestricted use of function symbols and extensions based in HILOG and Transaction Logic. Another possibility is to allow disjunction in the rule heads and introduce classical negation.

5.1 Basic WSML-Rule Syntax

5.2. WSML-Rule Ontologies

5.3. WSML-Rule Logical Expression Syntax

5.4. Goals in WSML-Rule

5.5. Mediators in WSML-Rule

5.6. Web Services in WSML-Rule

5.7. Differences between WSML-Flight and WSML-Rule

6. WSML-DL

WSML-DL will be an extension of WSML-Core to a full-fledged description logic with an expressiveness similar to OWL DL.

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

6.1 Basic WSML-DL Syntax

6.2. WSML-DL Ontologies

6.3. WSML-DL Logical Expression Syntax

6.4. Goals in WSML-DL

6.5. Mediators in WSML-DL

6.6. Web Services in WSML-DL

6.7. Differences between WSML-Code and WSML-DL

7. WSML-Full

WSML-Full combines First-Order Logic with nonmonotonic negation in order to provide a fully 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 umbrella.

WSML-Full is as yet underdefined, because the semantics of WSML-Full is not yet clear. There are several possible directions for the

7.1 Basic WSML-Full Syntax

7.2. WSML-Full Ontologies

7.3. WSML-Full Logical Expression Syntax

7.4. Goals in WSML-Full

7.5. Mediators in WSML-Full

7.6. Web Services in WSML-Full

7.7. 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.

7.8. Differences between WSML-Rule and WSML-Full

WSML-Full adds disjunction, classical negation, multiple model semantics.

PART III: THE WSML EXCHANGE SYNTAXES

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

The three exchange syntaxes for WSML are:

XML syntax:
: A syntax specifically tailored for machine processability, instead of human-readability; it is easily parsable by standard XML parsers, but is quite unreadable for humans. This syntax is defined in Chapter 8.

RDF syntax
: An alternate exchange syntax for WSML is WSML/RDF. WSML/RDF can be used to leverage the currently existing RDF tools, such a triple stores, and to syntactically combine WSML/RDF descriptions with other RDF descriptions. WSML/RDF is defined in Chapter 9.

Mapping to OWL
: A bidirectional mapping between (a subset of) OWL and WSML is given in Chapter 10.

8 XML Syntax for WSML

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

The complete XML Schema, including documentation, for the WSML/XML syntax can be found in Appendix B, along with an example of the use of this syntax. Future versions of this deliverable will contain a mapping from the human-readable syntax to the XML syntax, making the XML syntax easier to understand.

The basic namespace for WSML/XML is http://www.wsmo.org/2004/wsml. This is the namespace for all elements in WSML.

Future versions of this deliverable will contain a mapping between the human-readable syntax and the XML syntax.

The XML Schema for WSML and the documentation for the schema can be found in Appendix B.

It is good practice to use the xml:base attribute in the root element of any WSML specification. This xml:base should be made the same as the default namespace. This will allow generic XML applications to do URI resolution based on the definitions in the default namespace of the specification.

8.1. Future Work

After the XML Schema has been finalized, converters need to be built to convert from and to the human-readable syntax of WSML. The latter can be done using an XSLT script, as was done for previous versions (March 2004) of WSML/XML.

A mapping from the WSML human-readable syntax to the XML syntax have to be given inline in this section.

9 RDF Syntax for WSML

[JB: Notice that at this stage, this chapter only contains some general considerations concerning RDF and the suitability of the RDF data model for encoding ontologies]

In this chapter we will describe the RDF syntax for WSML and the relation between RDF meta-data and WSML ontologies. The Resource Description Framework RDF [RDF] is a framework for describing meta-data of resources on the Web. An RDF document consists of a number of triples. Each triple is of the form (*subject, predicate, object*). Each object can be the subject of another triple, yielding a directed labeled graph. Notice that a triple can be seen as a binary relation (denoted by the *predicate*) with the subject and the object of the triple as its domain.

An important property of RDF is that each resource (identified by a IRI) can play the role of a subject, predicate or object. Notice, however, that statements about predicates (i.e. statements with an identifier of a predicate as its subject) do not fall in the usual graph data model and thus it is hard to imagine what the meaning of such a statement is.

Another important property of RDF is that it allows for reification, i.e. statements about statements. Reification in RDF is achieved by designating a specific IRI as the identifier of a particular triple. Unfortunately, it is not possible to identify a statement directly. Instead, four additional triples are required in order to designate a IRI as the identifier of a particular triple. One triple is needed for each of the elements of the triple (subject, predicate, object). An additional triple is required to identify this resource as indeed an rdf:Statement.

After a resource has been designated as being the identifier of a statement, it is possible to use this resource as the subject or object of other statements. These statements are then statements about the original statement. Notice that such reified statements do not fall neatly in the graph data model. Instead, in the case of reification, one can imagine different graphs on different levels. The graph on the lowest level consists of the regular RDF graph, leaving out the reified statements. One level higher, there exists one graph for each statement which is reified. The four triples required to reify a statement can be seen as a link between a graph on the lower level and a graph one level higher. Notice however that because the values occurring in the objects of triples is not restricted, there can be interdependencies between graphs on different levels. However, we do not expect this to occur very often; we expect statements about statements to be rather simple. We believe that reified statements generally either express restrictions on the relation expressed with the original statement or be of the nature "*statement* isBelievedBy *person*". For an example graphical depiction of reification, see Figure 9.1.

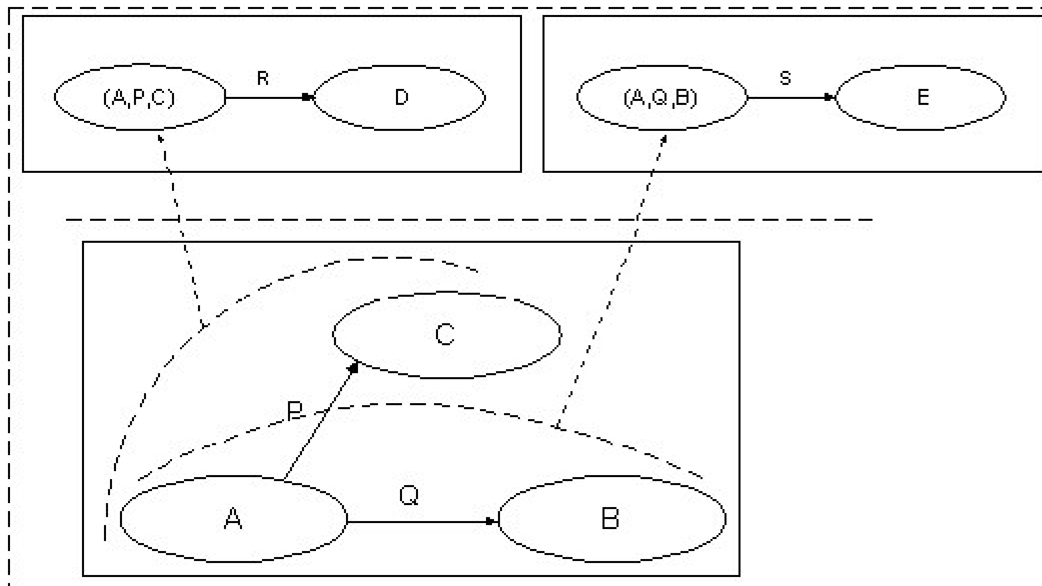

Figure 9.1. Using multiple graphs to represent reification in RDF

RDF has a number of designated predicates with a special meaning. The most important predicate is rdf:type. A statement with rdf:type as its predicate is interpreted such that the resource in the object of the triple indicates the *type* of the resource in the subject of the triple. Such typing triples are typically used to link resources to concepts in an ontology (where a concept is also seen as a resource), thus *rdf:type* corresponds with the *instance-of* relation common in ontology languages. In this case the *subject* of the triple is a member of the concept which plays the role of *object* in the triple. Furthermore, the predicate of a triple may refer to a an attribute specified in the ontology.

9.1 Representing Ontologies using RDF

Graphs have often been used to capture ontologies (e.g. [Mitra et al., 2000]) and RDF seems very suitable to capture ontology instance data, because of its graph-based data model. Relations between instances and relations between instances and concepts can be captured in RDF in a straightforward manner. However, when trying to capture ontologies in RDF, problems may arise because of incompatibility between the data model of RDF and the data model of the ontology language. In this section we describe for both RDFS and OWL (DL) how the RDF syntax relates to the ontology langauge.

RDFS [Brickley & Guha, 2004] has been developed as a lightweight ontology language specifically for describing RDF data. RDFS allows for the description of classes, class hierarchies, properties, property hierarchies and domain- and range restrictions for properties. RDFS itself is written down using RDF; an RDFS ontology solely consists of RDF triple. The *is-a* relationship which is used to construct class hierarchies is naturally expressed using a directed graph through the use of the predicate rdfs:subClassOf in triples, where rdfs:subClassOf corresponds with the *is-a*relationship. A property hierarchy is also naturally expressed using a graph through the predicate rdfs:subPropertyOf, where rdfs:subPropertyOf corresponds with the *is-a* relationship. Notice that these graphs are independent from each other.

A property in RDFS does not belong to a class. This is in contrast with databases and object-oriented programming, where properties are typically part of a class definition. It is, however, possible in RDFS to specify links between properties and classes through statements with the rdfs:domain and rdfs:range predicates. In such statements, the subject is a property and the object is a class. The use of such statements may lead to awkward

looking graphs (see Figure 9.2 for an example), because there are no references from the classes to the properties, but the other way around. Nonetheless, domain and range are naturally binary relations and are thus naturally expressed using triples. On the other hand, in case one wants to reuse the same property for different classes, one has to either omit the domain restriction, in which case the property can be used for all classes, or create several rdfs:domain statements, in which case any instance which uses this property must be an instance of all classes in the domain of this property, since multiple rdfs:domain statements are interpreted conjunctively.

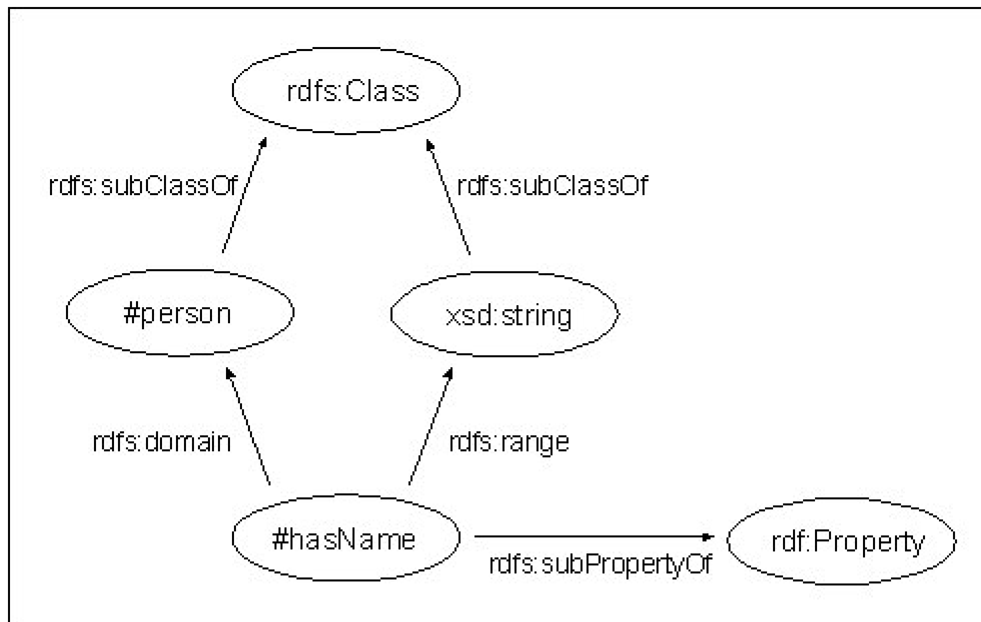

Figure 9.2. An example RDFS ontology in graph format

As we have seen above, RDFS ontologies can in most cases be captured in a graph-based data model, because in most cases the relations are binary. OWL [Dean & Schreiber, 2004] is a much more expressive ontology language than RDFS, allowing, among other things, local property restrictions. Such local restrictions involve a single class and are thus inherently relations with an arity higher than 2, because such a local restriction denotes the relation between a class, a property and the value of the restriction. For example, a cardinality restriction denotes the relationship between a class, a property and a number. This number denotes the cardinality. Such relations of arity higher than 2 lead to awkward constructions in the RDF syntax of OWL, because each relation needs to be written down using a number of RDF statements. Thus, we argue that the RDF data model is not suitable for encoding such more expressive ontology language constructs, because relations with an arity higher than 2 do not fit in the graph data model.

9.2 Representing WSML using RDF

Instead of encoding all ontology language constructs as RDF triples (as OWL does), we propose to follow the model of [Mitra et al., 2000]. There, the graph data model is only used to capture aspects of the ontology that naturally fit in the graph data model. Other aspects are encoded as additional rules which constrain the ontology; [Mitra et al., 2000] use Horn rules for the specification of these additional constraints.

We see three possible directions for the RDF syntax of WSML:

1. Following the style of RDFS and OWL by encoding all language constructs in RDF. The advantage(?) is that everything is encoded as RDF. The disadvantages are (1) that the RDF data model is largely ignored (i.e. this syntax does not allow you to

benefit from the RDF data model, which has simplicity as its major advantage), (2) that the data model of the RDF serialization does not correspond with the WSML data model; thus, reconstructing WSML from the RDF serialization is difficult and the relation between a particular RDF graph and the corresponding WSML ontology is not obvious (i.e. we have the same problem as OWL DL has with the mapping between the abstract syntax and the RDF syntax), and (3) the mentioned advantage that everything is encoded in RDF might even be a disadvantage; RDF tools might try to work with the artificial constructions introduced by the complex constructs of the language and can naturally not deal with them, because the data models do not match.

2. Encoding only the features of WSML which correspond naturally to the graph data model of RDF and represent more complex features which do not fit the graph model in an external document, which is ideally encoded using the XML syntax of WSML. The major disadvantage is that the constraints are not located in the same document as the RDF graph; thus, the ontology is split over several documents. A possible remedy would be to use snippets of WSML XML syntax in RDF triples using the rdf:XMLLiteral datatype.
3. Using the RDF reification mechanism to express n-ary relations. By using reification, it is possible to express properties of statements. Take for example the statement (A, hasAttribute, B). This statement associates an attribute with a class. A property of this statement could be a cardinality constraint, for example: ("A hasAttribute B", hasCardinality, 1). A major disadvantage of this approach is that these reified statements are not really statements about statements in the sense that in this example it is not the statement (A, hasAttribute, B) which has a cardinality of 1, but rather the attributes of the instances of A have this cardinality.

What we **do not consider as an option is to layer the RDF syntax of WSML on top of RDFS**. Such layering would suddenly redefine the semantics of the RDFS constructs and would clutter the syntax by mixing RDFS and WSML keywords. In our opinion it was a mistake in the design of OWL to include RDFS constructs in the OWL syntax. OWL (at least the Lite and DL variants) redefines the semantics of the RDFS constructs and mixes RDFS and OWL constructs in a non-intuitive manner: it is often not clear when modeling an OWL ontology whether the keyword should be taken from the RDFS or the OWL namespace. We do not intend to repeat this mistake in the design of the RDF syntax for WSML.

10 Mapping to OWL

This version of the deliverable contains only a mapping of WSML-Core to OWL. Once WSML-DL has been specified, a mapping of WSML-DL to OWL will be defined.

10.1. Mapping WSML-Core to OWL DL

In this chapter we define the semantics of WSML-Core through a mapping to the OWL DL abstract syntax [Patel-Schneider et al., 2004]. The subset of OWL DL which can be mapped to WSML-Core is OWL DL- and is described in [de Bruijn et al., 2004].

Table 10.1 shows the mapping between the WSML-Core conceptual syntax and OWL DL. In the table, *C* and *D* refer to named concepts (classes in OWL), *T* refers to a datatype, *R* refers to an ObjectProperty, *U* refers to a DatatypeProperty, *A* refers to an attribute (this can be either an ObjectProperty or a DatatypeProperty in OWL), *F* refers to a datatype predicate, *O* refers to an ontology and *M* refers to a mediator.

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

In Table 10.2, we have identified exactly which logical expressions can be translated to OWL DL. In the table, *C* and *D* refer to named concepts (classes in OWL), *T* refers to a datatype, *R* and *S* refer to ObjectProperties, *U* refers to a DatatypeProperty, *A* refers to an attribute (this can be either an ObjectProperty or a DatatypeProperty in OWL), and *F* refers to a datatype predicate. These logical expressions add little[3] in expressivity over the conceptual syntax, but sometimes the user prefers a different way of modeling axioms. Furthermore, this syntax for logical expressions can be directly used in goal descriptions and capability descriptions of web services for the modeling of assumptions, pre-conditions, effects and post-conditions. All and only those logical expressions that are either in the first column of Table 10.2 or can be reduced to logical expressions in Table 10.2 are valid in WSML-Core.

WSML-Core conceptual syntax	OWL DL Abstract syntax	Remarks
Logical Declarations		
ontology *O*	Ontology(*O*)	All definitions and axioms in the ontology are nested inside the Ontology statement, according to the other mappings in this table.
concept *C* **subConceptOf** {D_1,..., D_n} A_1 **impliesType** C_1 . . A_i **impliesType** C_n A_{i+1} **ofType** T_1	Class(*C* partial D_1 ... D_n restriction(A_1 allValuesFrom T_{i+1}) ... restriction(A_i allValuesFrom T_i) restriction(A_{i+1} allValuesFrom C_{i+1}) ...	Because of the strict separation between object-valued and data-values properties in OWL, it is required in OWL to specifically designate each property as DatatypeProperty or ObjectProperty. This is reflected in the translation.

Table 10.1: Mapping between WSML-Core and OWL DL abstract syntax

WSML-Core conceptual syntax	OWL DL Abstract syntax	Remarks
. . A_n **ofType** T_n	restriction(A_n allValuesFrom C_n))	
R **impliesType** [**transitive**] [**symmetric**] [**inverseOf**(R_0)] C	ObjectProperty(R [inverseOf(R_0)] [Symmetric] [Transitive])	In case the keyword **transitive** occurs in WSML-Core, the keyword "Transitive" occurs in OWL. Similar for **symmetric** and **inverseOf**.
relation R **subRelationOf** {R_1,..., R_n} [**domain ofType** C_1,..., C_n] [**range ofType** D_1,..., D_n]	ObjectProperty(R super(R_1) ... super(R_n) domain(C_1) ... domain(C_n) range(D_1) ... range(D_n)) DatatypeProperty(U super(U_1) ... super(U_n) domain(C_1) ... domain(C_n) range(T_1) ... range(T_n))	Notice that in case the range of a relation is a datatype, the relation is translated to a DatatypeProperty. Otherwise, it is translated to an ObjectProperty. The domain of a relation is not allowed to be a datatype.
datatype T [y **ofType** T_1] **definedBy** *deCom*	DatatypeExpression(T and(domain(T_1), *deCom*))	The keyword DatatypeExpression is not included in OWL, but is borrowed from OWL-E, which has a richer expressiveness for representing datatypes.
function F [**range ofType** T_1] [y_9 **ofType** T_9] . . [y_k **ofType** T_k] **definedBy** *deCom*	DatatypeExpression(F and(domain(T_1,...,T_n), *deCom*))	The keyword DatatypeExpression is not included in OWL, but is borrowed from OWL-E, which has a richer expressiveness for representing datatypes.
instance o **memberOf** C_1,..., C_n A_1 **hasValue** v_1 . . A_n **hasValue** v_n	Individual(o type(C_1) ... type(C_n) value(A_1 v_1) ... value(A_n v_n))	Notice that v_i can be either an instance of a concept or a data value (i.e. literal).
Extra-Logical Declarations		
nonFunctionalProperties P_1 v_1 . . P_n v_n [**version** v_0]	annotation(P_1 v_1) . . annotation(P_n v_n) [annotation(owl:versionInfo v_0)]	Annotation properties are nested inside other definitions, such as ontology, individual, class and property definitions. Notice, however, that when writing an annotations on the ontology level in OWL, they should be written with a capital 'A' (e.g. Annotation(owl:versionInfo "$Revision: 1.13 $")).
importedOntologies {	Annotation(owl:import O_1) .	

WSML-Core conceptual syntax	OWL DL Abstract syntax	Remarks
O_1, .., *O_n* }	. Annotation(owl:import *O_n*)	
usedMediators { *M_1*, .., *M_n* }		OWL does not have the concept of a mediator. Therefore, this construct cannot be translated to OWL.

Table 10.1: Mapping between WSML-Core and OWL DL abstract syntax

WSML-Core Logical Expression	OWL DL Abstract Syntax	Remarks
TODO: find a better way to characterize the allowed logical expressions		
C **subConceptOf** D_1, ..., D_n	Class (C partial D_1 ... D_n)	
?x **memberOf** D **->** ?x **memberOf** C	Class (D partial C)	
?x **memberOf** D **<->** ?x **memberOf** C_1 **and** ... **and** ?x **memberOf** C_n	Class (D complete C_1 ... C_n)	
C[U **ofType** T]	Class (C partial restriction(U allValuesFrom T))	
S(?x,?y) < R(?x,?y)	ObjectProperty (R super(S))	
U_9(?x,?y) < U_1(?x,?y)	DatatypeProperty (U_1 super(U_9))	
?x **memberOf** C **<-** R(?x,?y)	ObjectProperty (R domain(C))	
?y **memberOf** C **<-** R(?x,?y)	ObjectProperty (R range(S))	
S(?y,?x) **<-** R(?x,?y) R(?y,?x) **<-** S(?x,?y)	ObjectProperty (R inverseOf(S))	

Table 10.2: Mapping between WSML-Core logical expressions and OWL DL abstract syntax

WSML-Core Logical Expression	OWL DL Abstract Syntax	Remarks
R(?y,?x) **<-** R(?x,?y)	ObjectProperty (R Symmetric)	
R(?x,?y) **<-** R(?x,?y) **and** R(?y,?z)	ObjectProperty (R Transitive)	
o **memberOf** C [A_1 **hasValue** o_1, . . A_n **hasValue** o_n]	Individual(o type(C) value(A_1 o_1) ... value(A_n o_n))	Both the **memberOf** C and all the property values are optional.
Datatype expressions (mapping to OWL-E)		
$F_1(y_1,...,y_k)$ **and** ... **and** $F_n(y_1,...,y_k)$	and(F_1,...,F_n)	F_i stands for a datatype expression component, which in this case amounts to a datatype predicate of arity *k*.
deCom(y_1,...,y_k)	*deCom*	*deCom* stands for a datatype predicate of arity *k*.

Table 10.2: Mapping between WSML-Core logical expressions and OWL DL abstract syntax

10.2. Mapping OWL DL to WSML-Core

Table 10.3 shows the mapping between OWL DL (abstract syntax) and WSML-Core. It contains the conceptual syntax as well as any additional logical expression that might be necessary to capture the semantics of the OWL DL statement. The table shows for each construct supported by OWL DL the WSML-Core syntax in terms of the conceptual model and additional logical expressions necessary to capture the construct.

As we can see from Table 10.3, a large part of OWL DL can be captured in WSML-Core. However, only ontologies which fit in OWL DL- can be translated to WSML-Core Most of the axioms can be captured with the conceptual model. However, some axioms rely on additional logical expressions. Notice that if both conceptual syntax and a logical expression is given in the table, this means that they are both introduced in the translation and should be taken in conjunction.

OWL DL Abstract syntax	WSML-Core conceptual syntax	WSML-Core logical expression
TODO: use a recursive function for the translation		
Logical Definitions		
Class(*C* partial D_1 ... D_n)	**concept** *C*	?x **memberOf** C_1 **and** ... **and** ?x **memberOf** C_n

Table 10.3: Mapping between OWL DL abstract syntax and WSML-Core

OWL DL Abstract syntax	WSML-Core conceptual syntax	WSML-Core logical expression
TODO: use a recursive function for the translation		
	subConceptOf D_i (in case D_i is a named class) R_i **impliesType** P_i (in case D_i is a value restriction of the form R_i allValuesFrom A_i) U_i **ofType** P_i (in case D_i is a value restriction of the form U_i allValuesFrom T_i)	<- ?x **memberOf** C (in case D_i is of the form intersectionOf(C_1 ... C_n)
Class(C complete D_1 ... D_n)	**concept** C **subConceptOf** D_1, ..., D_n	?x **memberOf** C **<-** ?x **memberOf** D_1 **and** ... **and** ?x **memberOf** D_n
EquivalentClasses(C_1 ... C_n)	**concept** C_i **subConceptOf** C_j	
ObjectProperty(R super(R_1) ... super(R_n) domain(C_1) ... domain(C_n) range(D_1) ... range(D_n)	**relation** R **subRelationOf** R_1, ..., R_n **domain impliesType** { C_1, ..., C_n } **range impliesType** { D_1, ..., D_n }	
[inverseOf(R_0)]		R(?y,?x) **impliedBy** R_0(?x,?y). R_0(?y,?x) **impliedBy** R(?x,?y).
[Symmetric]		R(?y,?x) **impliedBy** R(?x,?y).
[Transitive])		R(?x,?z) **impliedBy** R(?x,?y), R(?y,?z).
SubPropertyOf(R_1 R_x)		R_9(?x,?y) **<-** R_1(?x,?y)
EquivalentProperties(R_1 ... R_n)		R_i(?x,?y) **<-** R_j(?x,?y)
DatatypeProperty(U super(U_1) ... super(U_n) domain(C_1) ... domain(C_n) range(T_1) ... range(T_n)	**relation** U **subRelationOf** U_1, ..., U_n **domain impliesType** { C_1, ..., C_n } **range ofType** { T_1, ..., T_n }	
SubPropertyOf(U_1 U_x)		U_9(?x,?y) **<-** U_1(?x,?y)

Table 10.3: Mapping between OWL DL abstract syntax and WSML-Core

OWL DL Abstract syntax	**WSML-Core conceptual syntax**	**WSML-Core logical expression**
TODO: use a recursive function for the translation		
EquivalentProperties(U_1 ... U_n)		U_i(?x,?y) **<-** U_j(?x,?y)
Individual(o type(C_1) ... type(C_n) value(R_1 o_1) ... value(R_n o_n) value(U_1 v_1) ... value(U_n v_n))	**instance** o **memberOf** C_1,..., C_n R_1 **hasValue** o_1 . . R_n **hasValue** o_n U_1 **hasValue** v_1 . . U_n **hasValue** v_n	
DatatypeExpression(P *deCom*)	**datatype** P **definedBy** *deCom* (in case *deCom* is a unary predicate or a conjunction of unary predicates)	
domain(T_1 ... T_k)	?value **ofType** T_1 (in case $k = 1$) $?y_1$ **ofType** T_1 . . $?y_n$ **ofType** T_k (in case $k > 1$)	
and(F_1 ... F_n)		$F_1(?y_1,...,?y_k)$ **and** ... **and** $F_n(?y_1,...,?y_k)$
or(F_1 ... F_n)		$F_1(?y_1,...,?y_k)$ **or** ... **or** $F_n(?y_1,...,?y_k)$
Extra-Logical Definitions		
annotation(P v)	**nonFunctionalProperties** P **hasValue** v	

Table 10.3: Mapping between OWL DL abstract syntax and WSML-Core

PART IV: CONCLUSIONS

11. Related Efforts

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

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

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

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

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

12. Conclusions

References

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

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

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

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

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

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

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

[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., 2004] J. de Bruijn, A. Polleres, R. Lara and D. Fensel. *OWL⁻*. Deliverable d20.1v0.2, WSML, 2004. http://www.wsmo.org/2004/d20/d20.1/v0.2/

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

[Duerst & Suignard, 2004] M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. IETF Last Call draft 27 September 2004. http://www.w3.org/International/iri-edit/draft-duerst-iri-10.txt

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

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

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

[Eiter et al., 2004] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proc. of the*

International Conference of Knowledge Representation and Reasoning (KR04), 2004.

[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.

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

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

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

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

[Lara et al., 2004] R. Lara, D. Roman, A. Polleres, and D. Fensel. A Conceptual Comparison of WSMO and OWL-S. In *European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, 2004.

[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.

[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/.

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

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

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

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

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

[Roman et al., 2004] D. Roman, H. Lausen, and U. Keller (eds.): *Web Service Modeling Ontology - Standard (WSMO - Standard)*, WSMO deliverable D2 version 1.0. available from http://www.wsmo.org/2004/d2/v1.0/.

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

[Stollberg et al., 2004] M. Stollberg, H. Lausen, A. Polleres, and R. Lara (eds.): *WSMO Use Case Modeling and Testing*, WSMO d3.2v0.1, version 2004-06-28. available from http://www.wsmo.org/2004/d3/d3.2/v0.1/20040628/.

[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, 2001] G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the semantic web. *Journal on Data Semantics*, 1:69-97, 20031.

Acknowledgement

We would especially like to thank the reviewer of the deliverable, Ian Horrocks, for useful comments and discussions around this deliverable.

We would like to acknowledge Michael Felderer and Reto Krummenacher for their work on the grammar.

The work is funded by the European Commission under the projects DIP, Knowledge Web, SEKT, SWWS, Esperonto, and h-TechSight; by Science Foundation Ireland under the DERI-Lion project; and by the Vienna city government under the CoOperate program.

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.

Appendix A. Grammar for the human-readable syntax

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

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

A.1. BNF-style grammar for WSML

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_ncnamechar    = letter | digit | '_' | combiningchar | extender
escaped_ncnamechar        = '.' | '-' | not_escaped_ncnamechar
ncnamechar                = ( escape_char escaped_ncnamechar ) | not_escaped_ncnamechar
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                          = scheme ':' ihier_part ( '?' iquery )? ( '#' ifragment )?
absolute_iri                 = scheme ':' ihier_part ( '?' iquery )?
relative_iri                 = ihier_part ( '?' iquery )? ( '#' ifragment )?
iric                         = reserved | iunreserved | escaped
iri_reference                = iri | relative_iri
tab                          = 9
cr                           = 13
lf                           = 10
eol                          = cr lf | cr | lf
squote                       = '''
dquote                       = '"'
not_cr_lf                    = [ all - [ cr+ lf ] ]
escaped_char                 = escape_char all
not_escape_char_not_dquote   = [ all - [ '"' + escape_char ] ]
not_escape_char_not_squote   = [ all - [ ''' + escape_char ] ]
literal_content              = escaped_char | not_escape_char_not_dquote
string_content               = escaped_char | not_escape_char_not_squote
not_star                     = [ all - '*' ]
not_star_slash               = [ not_star - '/' ]
long_comment                 = '/*' not_star* '*'+ ( not_star_slash not_star* '*'+ )* '/'
begin_comment                = '//' | 'comment'
short_comment                = begin_comment not_cr_lf* eol
comment                      = short_comment | long_comment
blank                        = ( ' ' | tab | eol )+
qmark                        = '?'
luridel                      = '<"'
ruridel                      = '">'
primary_subtag               = letter+
language_subtag              = ( letter | digit )+
language_tag                 = '@' primary_subtag ( '-' language_subtag )*
```

Tokens

```
t_blank      = blank
t_comment    = comment
comma        = ','
endpoint     = '.' blank
pathcon      = '.'
dblcaret     = '^^'
lpar         = '('
rpar         = ')'
lbracket     = '['
rbracket     = ']'
lbrace       = '{'
rbrace       = '}'
colon        = ':'
and          = 'and'
```

```
or                  = 'or'
implies             = 'implies'  |  '->'
implied_by          = 'impliedBy'  |  '<-'
equivalent          = 'equivalent'  |  '<->'
not                 = 'neg'  |  'naf'
exists              = 'exists'
forall              = 'forAll'
univ_false          = 'false'
univ_true           = 'true'
gt                  = '>'
lt                  = '<'
gte                 = '>='
lte                 = '=<'
equals              = '='
unequal             = '!='
add_op              = '+'
sub_op              = '-'
star                = '*'
div_op              = '/'
t_wsmlvariant       = 'wsmlVariant'
t_namespace         = 'namespace'
t_targetnamespace   = 'targetNamespace'
t_nfp               = 'nonFunctionalProperties'  |  'nfp'
t_endnfp            = 'endNonFunctionalProperties'  |  'endnfp'
t_importontology    = 'importedOntologies'
t_usemediator       = 'usedMediators'
t_oomediator        = 'ooMediator'
t_ggmediator        = 'ggMediator'
t_wgmediator        = 'wgMediator'
t_wwmediator        = 'wwMediator'
t_source            = 'source'
t_target            = 'target'
t_useservice        = 'useService'
t_goal              = 'goal'
t_webservice        = 'webService'
t_use_capability    = 'useCapability'
t_capability        = 'capability'
t_precondition      = 'precondition'
t_postcondition     = 'postcondition'
t_assumption        = 'assumption'
t_effect            = 'effect'
t_use_interface     = 'useInterface'
t_interface         = 'interface'
t_choreography      = 'choreography'
t_orchestration     = 'orchestration'
t_ontology          = 'ontology'
t_concept           = 'concept'
t_subconcept        = 'subConceptOf'
t_oftype            = 'ofType'
t_impliestype       = 'impliesType'
t_relation          = 'relation'
t_subrelation       = 'subRelationOf'
t_function          = 'function'
t_range             = 'range'
t_instance          = 'instance'
t_memberof          = 'memberOf'
```

```
t_hasvalue          = 'hasValue'
t_datatype          = 'datatype'
t_axiom             = 'axiom'
t_relation_instance = 'relationInstance'
t_definedby         = 'definedBy'
t_transitive        = 'transitive'
t_symmetric         = 'symmetric'
t_inverseof         = 'inverseOf'
t_reflexive         = 'reflexive'
variable            = qmark alphanum+
anonymous           = '_#' digit*
pos_int             = digit+
pos_float           = digit+ '.' digit+
string              = squote string_content* squote
plainliteral        = dquote literal_content* dquote ( language_tag )?
full_iri            = luridel iri_reference ruridel
ncname              = ( letter | '_' ) ncnamechar*
```

Ignored Tokens

- t_blank
- t_comment

Productions

```
wsml                    = wsmlvariant? namespace? targetnamespace? definition*
wsmlvariant             = t_wsmlvariant full_iri
namespace               = t_namespace prefixdefinitionlist
prefixdefinitionlist    = {defaultns}            full_iri
                        | {prefixdefinitionlist} lbrace prefixdefinition moreprefixdefinitions* rbrace
prefixdefinition        = {namespacedef} ncname full_iri
                        | {default}      full_iri
moreprefixdefinitions   = comma prefixdefinition
targetnamespace         = t_targetnamespace full_iri
definition              = {goal}       goal
                        | {ontology}   ontology
                        | {webservice} webservice
                        | {mediator}   mediator
header                  = {nfp}                 nfp
                        | {usedmediators}       usedmediators
                        | {importedontologies}  importedontologies
usedmediators           = t_usemediator idlist
importedontologies      = t_importontology idlist
nfp                     = t_nfp attributevalue* t_endnfp
mediator                = {oomediator} oomediator
                        | {ggmediator} ggmediator
                        | {wgmediator} wgmediator
                        | {wwmediator} wwmediator
oomediator              = t_oomediator id? nfp? importedontologies? source* target? use_service?
ggmediator              = t_ggmediator id? header* source* target? use_service?
wgmediator              = t_wgmediator id? header* source? target? use_service?
```

```
wwmediator             = t_wwmediator id? header* source? target? use_service?
use_service            = t_useservice id
source                 = t_source id
target                 = t_target id
goal                   = t_goal id? nfp? usedmediators? importedontologies* postcondition_or_effect*
postcondition_or_effect = {postcondition} t_postcondition axiomdefinition
                       | {effect}         t_effect axiomdefinition
webservice             = t_webservice id? nfp? usedmediators? importedontologies* capability?
                         interface*
capability             = {use_capability}     t_use_capability id
                       | {defined_capability} capabilitydef
capabilitydef          = t_capability id? nfp? usedmediators? importedontologies?
                         pre_post_ass_or_eff*
pre_post_ass_or_eff    = {precondition}   t_precondition axiomdefinition
                       | {assumption}     t_assumption axiomdefinition
                       | {post_or_effect} postcondition_or_effect
interface              = {use_interface}     t_use_interface id
                       | {defined_interface} interfacedef
interfacedef           = t_interface id? nfp? usedmediators? importedontologies? choreography?
                         orchestration?
choreography           = t_choreography id
orchestration          = t_orchestration idlist
ontology               = t_ontology id? header* ontology_element*
ontology_element       = {concept}          concept
                       | {instance}         instance
                       | {relation}         relation
                       | {function}         function
                       | {relationinstance} relationinstance
                       | {axiom}            axiom
                       | {datatype}         datatype
concept                = t_concept id superconcept? nfp? attribute* log_definition?
superconcept           = t_subconcept idlist
att_type               = {constraint} t_oftype
                       | {restriction} t_impliestype
attribute              = [attr]: id att_type cardinality? attributefeature* [type]: id nfp?
cardinality            = lpar [min_cardinality]: pos_int [max_cardinality]: cardinality_number? rpar
cardinality_number     = {finite_cardinality}   pos_int
                       | {infinite_cardinality} star
attributefeature       = {transitive} t_transitive
                       | {symmetric}  t_symmetric
                       | {inverse}    t_inverseof lpar id rpar
                       | {reflexive}  t_reflexive
instance               = t_instance id? memberof? nfp? attributevalue*
memberof               = t_memberof idlist
attributevalue         = id t_hasvalue idlist nfp?
relation               = t_relation id superrelation? nfp? parameter* log_definition?
superrelation          = t_subrelation idlist
parameter              = [parameter]: id att_type [type]: id
```

```
function             =   t_function id superrelation? nfp? functionparameters? log_definition?
functionparameters   =   parameter* t_range att_type id
relationinstance     =   t_relation_instance [relation]: id? t_memberof [type]: id nfp? attributevalue*
datatype             =   t_datatype axiomdefinition
axiom                =   t_axiom axiomdefinition
axiomdefinition      =   {use_axiom}     id
                     |   {defined_axiom} id? nfp? log_definition
log_definition       =   t_definedby log_expr
log_expr             =   expr endpoint
expr                 =   {implication} expr imply_op complex
                     |   {complex}     complex
complex              =   {quantified}  quantified
                     |   {disjunction} disjunction
quantified           =   quantifier_key variablelist lpar expr rpar
disjunction          =   {conjunction} conjunction
                     |                 disjunction or conjunction
conjunction          =   {subexpr} subexpr
                     |             conjunction and subexpr
subexpr              =   {negated} not subexpr
                     |   {simple}  simple
                     |             lpar expr rpar
simple               =   {molecule}    molecule
                     |   {composition} composition
                     |   {atom}        term
molecule             =   {concept_molecule}   concept_molecule
                     |   {attribute_molecule} attr_molecule
concept_molecule     =                        [left]: term cpt_op [right]: term
                     |   {attr_cpt_molecule} attr_molecule cpt_op term
                     |   {cpt_attr_molecule} term cpt_op attr_molecule
attr_molecule        =   term lbracket attr_rel_list rbracket
attr_rel_list        =   {attr_relation} attr_relation
                     |                   attr_rel_list comma attr_relation
attr_relation        =   [left]: term attr_op [right]: term
composition          =   [left]: term comp_op [right]: term
functionsymbol       =   {empty}         term lpar rpar
                     |   {parametrized}  term lpar termlist rpar
termlist             =   {term} term
                     |          termlist comma term
term                 =   {constant} id
                     |   {math}     lpar mathexpr math_op term rpar
                     |   {function} functionsymbol
                     |   {pathconn} term pathcon id
mathexpr             =   {sub} mathexpr math_op term
                     |         term
variablelist         =   {variable}      variable
                     |   {variable_list} variablelist comma variable
```

```
comp_op          =   {gt}        gt
                   | {lt}        lt
                   | {gte}       gte
                   | {lte}       lte
                   | {equal}     equals
                   | {unequal}   unequal
cpt_op           =   {memberof}     t_memberof
                   | {subconceptof} t_subconcept
quantifier_key   =   {forall} forall
                   | {exists} exists
attr_op          =   {oftype}       t_oftype
                   | {impliestype}  t_impliestype
                   | {hasvalue}     t_hasvalue
imply_op         =   {implies}     implies
                   | {impliedby}   implied_by
                   | {equivalent}  equivalent
math_op          =   {add} add_op
                   | {sub} sub_op
                   | {mul} star
                   | {div} div_op
moreids          =   comma id
qname            =   {any}      prefix? ncname
                   | {relation} prefix  t_relation
                   | {source}   prefix  t_source
prefix           =   ncname colon
id               =   {iri}                  full_iri
                   | {qname}                qname
                   | {literal}              literal
                   | {anonymous}            anonymous
                   | {var}                  variable
                   | {universal_truth}      univ_true
                   | {universal_falsehood}  univ_false
idlist           =   {id}     id
                   | {idlist} lbrace id moreids* rbrace
neg_int          =   sub_op pos_int
float            =   sub_op? pos_float
number           =   {positive_int} pos_int
                   | {negative_int} neg_int
                   | {float}        float
literal          =   {typedliteral} typedliteral
                   | {plainliteral} plainliteral
                   | {numeric}      number
                   | {string}       string
typedliteral     =   plainliteral dblcaret id
```

A.2. An example of WSML in the human-readable syntax

```
wsmlVariant _"http://www.wsmo.org/2004/wsml/wsml-flight"

namespace {_"http://www.example.org/ontologies/example#",
    dc     _"http://purl.org/dc/elements/1.1#",
    foaf _"http://xmlns.com/foaf/0.1/",
```

```
    xsd      <"http://www.w3.org/2001/XMLSchema#">,
    wsml <"http://www.wsmo.org/2004/wsml#">,
    loc      <"http://www.wsmo.org/ontologies/location#">,
    oo       <"http://example.org/ooMediator#">}

targetNamespace <"http://www.example.org/ontologies/example#">

/*****************************
 * ONTOLOGY
 *****************************/
ontology
    nfp
        dc:title hasValue 'WSML example ontology'
        dc:subject hasValue 'family'
        dc:description hasValue 'fragments of a family ontology to provide WSML examples'
        dc:contributor hasValue { <"http://homepage.uibk.ac.at/~c703240/foaf.rdf">,
            <"http://homepage.uibk.ac.at/~csaa5569/">,
            <"http://homepage.uibk.ac.at/~c703239/foaf.rdf">,
            <"http://homepage.uibk.ac.at/homepage/~c703319/foaf.rdf"> }
        dc:date hasValue "2004-11-22"^^xsd:date
        dc:format hasValue 'text/html'
        dc:language hasValue 'en-US'
        dc:rights hasValue <"http://www.deri.org/privacy.html">
        wsml:version hasValue '$Revision: 1.32 $'
    endnfp

    usedMediators { <"http://example.org/ooMediator"> }

    importedOntologies { <"http://www.wsmo.org/ontologies/location">,
        <"http://xmlns.com/foaf/0.1"> }

    /*
     * This Concept illustrates the use of different styles of
     * attributes.
     */
    concept Human
        nonFunctionalProperties
            dc:description hasValue 'concept of a human being'
        endNonFunctionalProperties
        hasName ofType foaf:name
        hasAge ofType (1) xsd:integer
        hasParent impliesType inverseOf(hasChild) Human
        hasChild impliesType Human
        hasAncestor impliesType transitive Human
        hasWeight ofType (1) xsd:float
        hasBirthdate ofType (1) xsd:date
        hasObit ofType (0 1) xsd:date
        isAlive ofType (1) xsd:boolean
        hasBirthplace ofType (1) loc:location
        isMarriedTo impliesType (0 1) symmetric Human
        hasCitizenship ofType oo:country

    /*
     * This axiom shows the usage of built-in predicates
     * (based on the XQuery Built-Ins)
     */
    axiom AgeOfHuman
        nonFunctionalProperties
            dc:description hasValue 'Axiom defining the hasAge attribute of a human
                given its birthdate.'
        endNonFunctionalProperties
        definedBy
            forAll ?x,?y,?z (
                ?x[hasAge hasValue ?y] impliedBy
                    ?x memberOf Human and
                    wsml:yearsFromDuration(?y, ?z) and
                    wsml:subtractDateTimesYieldingDayTimeDuration(
                            ?z,
                            ?x.hasBirthdate,
                            wsml:currentDateTime())
            ) .

    /*
     * This axiom shows that a domain ontology can hide complex
     * relations and ease the definition of goals subsequently.
     */
     axiom IsAlive
         definedBy
             forAll ?x, ?y(
                 ?x.isAlive = ?y impliedBy
                     ?x memberOf Human and
```

```
                    ((?y = true) or
                        (?y = false and wsml:beforeDate(?x.hasObit, wsml:currentDate()))
                    )
          ).
/*
 * Illustrating general disjointness between two classes
 * via a constraint
 */
concept Man subConceptOf Human
concept Woman subConceptOf Human
axiom ManDisjointWoman
    definedBy
        false impliedBy ?x memberOf Man and ?x memberOf Woman.

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

/*
 * Using a definedBy to define class membership and an additional
 * axiom as constraint
 */
concept Child subConceptOf Human
    nfp
        dc:description hasValue 'Human being not older than 14'
    endnfp
    definedBy
        forAll ?x (
            ?x memberOf Human and ?x.age =<14 implies ?x memberOf Child).

axiom ValidChild
    nfp
        dc:description hasValue 'Note: ?x.age > 14      would imply that the
            constraint is violated if the age is known to be bigger than 14;
            the chosen axiom neg ?x.age =< 14 on the other hand says that
            whenever you know the age and it is lessor  equal 14 the constraint
            is not violated, i.e. if the age is not  given the constraint is violated.'
    endnfp
    definedBy
        false impliedBy ?x memberOf Child and neg ?x.age =< 14 .

/*
 * Defining complete subclasses by use of axioms
 */
concept Girl subConceptOf Woman
concept Boy

/*
 * This axiom implies that Boy is a subconcept of Man and Child
 */
axiom ABoy
    definedBy
        forAll ?x (
            ?x memberOf Boy equivalent ?x memberOf Man and ?x memberOf Child ) .

/*
 * This axioms implies that every child is either a boy or a girl.
 * That is not the same as the axiom ManDisjointWoman, that simply says that
 * one cannot be man and woman at once. However it would be generally reasonable
 * to combine the two and to provide an axiom CompletenessOfHumans.
 */
axiom CompletenessOfChildren
    definedBy
        forAll ?x (
            ?x memberOf Child equivalent ?x memberOf Girl or ?x memberOf Boy ) .

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

instance Paul memberOf {Parent, Man}
    hasName hasValue 'Paul Smith'
```

```
        hasBirthdate hasValue "1976-08-16"^^xsd:date
        hasChild hasValue George
        hasCitizenship hasValue oo:de

    instance Susan memberOf Woman
        hasName hasValue 'Susan Jones'
        hasBirthdate hasValue "1976-08-16"^^xsd:date

     /*
        * This will be automatically an instance of Boy, as that Man is younger than 14.
        */
    instance George memberOf Man
        hasName hasValue "George Smith"^^xsd:string
        hasAncestor hasValue Mary
        hasWeight hasValue "3.52"^^xsd:float
        hasBirthdate hasValue "2004-10-21"^^xsd:date

    /*
     * Custom datatype predicate (with named parameters)
     */
    function bodyMassIndex
        nfp
            dc:description hasValue 'calculates the Body Mass Index
                given by bmi = kg/m2.'
        endnfp
        length ofType xsd:float
        weight ofType xsd:integer
        range ofType xsd:float
        definedBy
            bodyMassIndex(length hasValue ?len,
                                        weight hasValue ?wei,
                                        range hasValue ?bmi)
            equivalent ?bmi = (?wei/(?len*?len)) .



/*****************************
 * WEBSERVICE
 *****************************/
webService <"http://example.org/Germany/BirthRegistration">
    nfp
        dc:title hasValue 'Birth registration service for Germany'
        dc:type hasValue <"http://www.wsmo.org/2004/d2/#webservice">
        wsml:version hasValue '$Revision: 1.32 $'
    endnfp

    usedMediators { <"http://example.org/ooMediator"> }

    importedOntologies { <"http://www.example.org/ontologies/example">,
        <"http://www.wsmo.org/ontologies/location"> }

    capability
        precondition
            nonFunctionalProperties
                dc:description hasValue 'The input has to be boy or a girl
                    with birthdate in the past and be born in Germany.'
            endNonFunctionalProperties
            definedBy
                (?child memberOf Child)
                and wsml:dateLessThan(?child.hasBirthdate,wsml:currentDate())
                and ( ?child.hasBirthplace.locatedIn = oo:de
                    or ?child.hasParent.hasCitizenship = oo:de ) .

        assumption
            nonFunctionalProperties
                dc:description hasValue 'The child is not dead'
            endNonFunctionalProperties
            definedBy
                (?child memberOf Child)
                    and neg (exists ?x (?child.hasObit = ?x)).

        effect
            nonFunctionalProperties
                dc:description hasValue 'After the registration the child
                    is a German citizen'
            endNonFunctionalProperties
            definedBy
                ?child memberOf Child
                    and ?child.hasCitizenship = oo:de.
```

```
    interface
        choreography <"http://example.org/tobedone">
        orchestration <"http://example.org/tobedone">


/******************************
 * GOAL
 ******************************/
goal <"http://example.org/Germany/GetCitizenShip">
    nonFunctionalProperties
        dc:title hasValue "Goal of getting a citizenship within Germany"^^xsd:string
        dc:type hasValue <"http://www.wsmo.org/2004/d2#goals">
        wsml:version hasValue '$Revision: 1.32 $'
    endNonFunctionalProperties

    usedMediators { <"http://example.org/ooMediator"> }

    importedOntologies { <"http://www.example.org/ontologies/example">,
        <"http://www.wsmo.org/ontologies/location"> }

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

goal <"http://example.org/Germany/RegisterGeorge">
    nfp
        dc:title hasValue 'Goal of getting a Registration for Paul\'s son George'
        dc:type hasValue <"http://www.wsmo.org/2004/d2#goals">
        wsml:version hasValue '$Revision: 1.32 $'
    endnfp

    usedMediators { <"http://example.org/ooMediator"> }

    importedOntologies      { <"http://www.example.org/ontologies/example">,
        <"http://www.wsmo.org/ontologies/location"> }

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


/******************************
 * MEDIATOR
 ******************************/
ooMediator <"http://example.org/ooMediator">
    nonFunctionalProperties
        dc:description hasValue 'This ooMediator translates the owl
            description of the iso ontology to wsml and adds the
            necessary statements to make them memberOf loc:country
            concept of the wsmo location ontology.'
        dc:type hasValue <"http://www.wsmo.org/2004/d2/#ggMediator">
        wsml:version hasValue '$Revision: 1.32 $'
    endNonFunctionalProperties
    source <"http://www.daml.org/2001/09/countries/iso#">
    source <"http://www.wsmo.org/ontologies/location">

/*
 * This mediator is used to link the two goals. The mediator defines
 * a connection between the general goal ('GetCitizenShip') as
 * generic and reusable goal which is refined in the concrete
 * goal ('RegisterGeorge').
 */
ggMediator <"http://example.org/ggMediator">
    nonFunctionalProperties
        dc:title hasValue 'GG Mediator that links the general goal of getting a citizenship
                                            with the concrete goal of registering George'
        dc:subject hasValue { 'ggMediator', 'Birth', 'Online Birth-Registration' }
        dc:type hasValue <"http://www.wsmo.org/2004/d2/#ggMediator">
        wsml:version hasValue '$Revision: 1.32 $'
    endNonFunctionalProperties
    source <"http://example.org/GetCitizenShip">
    target <"http://example.org/RegisterGeorge">
```

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

Appendix B. XML Schemas for the XML exchange syntax

In the following sections we present the XML Schemas for the XML syntax of WSML, as well as an example. This example is the XML version of the example of Section A.1 [Notice that at the moment these are not completely in-sync and there might be discrepancies between the two examples].

Appendix B.1. XML Schema for the XML syntax of WSML

The following is an XML schema of WSML. The schema is also available online at http://www.wsmo.org/2004/d16/d16.1/v0.2/xml-syntax/wsml-xml-syntax.xsd.

This schema includes two module schemas. One for the WSML identifiers (http://www.wsmo.org/2004/d16/d16.1/v0.2/xml-syntax/wsml-identifiers.xsd) and one for the logical expressions of WSML (http://www.wsmo.org/2004/d16/d16.1/v0.2/xml-syntax/wsml-expr.xsd). Furthermore, the schema imports an additional schema for the basic Dublin Core elements (http://dublincore.org/schemas/xmls/qdc/2003/04/02/dc.xsd).

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

XML syntax for WSML:

http://www.wsmo.org/2004/d16/d16.1/v0.2/xml-syntax/documentation/wsml-xml-syntax.xsd.html

XML syntax for WSML identifiers:

http://www.wsmo.org/2004/d16/d16.1/v0.2/xml-syntax/documentation/wsml-identifiers.xsd.html

XML syntax for WSML logical expressions:

http://www.wsmo.org/2004/d16/d16.1/v0.2/xml-syntax/documentation/wsml-expr.xsd.html

Appendix B.2. An example of a WSML/XML ontology

An example WSML/XML specification, which roughly corresponds to the example of Appendix A.1 can be found at: http://www.wsmo.org/2004/d16/d16.1/v0.2/xml-syntax/wsml-testing.xml.

Appendix C. RDF Schemas for the RDF exchange syntax

This section will contain the RDF Schema definitions for the RDF syntax for all the WSML variants.

Appendix D. Built-ins in WSML

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

Appendix D.1. WSML Datatype predicates

This section contains a list of datatype predicates suggested for use in WSML. These predicates correspond to functions in XQuery/XPath [Malhotra et al., 2004]. Notice that SWRL [Horrocks et al., 2004] built-ins support is also based on XQuery/XPath.

The current list is only based on the built-in support in the WSML language through the use of special symbols. Find a translation of the built-in symbols to datatype predicates in the next section. The symbol 'range' signifies the range of the function. Functions in XQuery have a defined range, whereas predicates only have a domain. Therefore, the first argument of a WSML datatype predicate which represents a function represents the range of the function. Comparators in XQuery are functions, which return a boolean value. These comparators are directly translated to predicates. If the XQuery function returns 'true', the arguments of the predicate are in the extension of the predicate. See Table D.1 for the complete list. In the evaluation of the predicates, the parameters 'A' and 'B' must be bound; the parameter 'range' does not need to be bound. A parameter is bound if it is substituted with a value. It is not bound if it is substituted with a variable. The variable is then used to convey some outcome of the function.

WSML datatype predicate	XQuery function	Datatype (A)	Datatype (B)	Return datatype
wsml:numeric-equal(A,B)	op:numeric-equal(A,B)	numeric	numeric	
wsml:numeric-greater-than(A,B)	op:numeric-greater-than(A,B)	numeric	numeric	
wsml:numeric-less-than(A,B)	op:numeric-less-than(A,B)	numeric	numeric	
wsml:string-equal(A,B)	op:numeric-equal(fn:compare(A, B), 1)	xsd:string	xsd:string	
wsml:numeric-add(range,A,B)	op:numeric-add(A,B)	numeric	numeric	numeric
wsml:numeric-subtract(range,A,B)	op:numeric-subtract(A,B)	numeric	numeric	numeric
wsml:numeric-multiply(range,A,B)	op:numeric-multiply(A,B)	numeric	numeric	numeric
wsml:numeric-divide(range,A,B)	op:numeric-divide(A,B)	numeric	numeric	numeric

Table D.1: WSML Datatype Predicates

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

Appendix D.2. Translating built-in symbols to Datatype predicates

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

We distinguish between built-in functions and built-in relations. Functions have a defined domain and range. Relations only have a domain and can in fact be seen as functions,

which return a boolean, as in XPath/XQuery [Malhotra et al., 2004]. We first provide the translation of the built-in relations and then present the rewriting rules for the built-in functions.

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

Operator	Datatype (A)	Datatype (B)	Predicate
A = B	xsd:string	xsd:string	wsml:string-equal(A,B)
A != B	xsd:string	xsd:string	wsml:not(wsml:string-equal(A,B))
A = B	numeric	numeric	wsml:numeric-equal(A,B)
A != B	numeric	numeric	wsml:not(wsml:numeric-equal(A,B))
A < B	numeric	numeric	wsml:less-than(A,B)
A =< B	numeric	numeric	wsml:less-equal(A,B)
A > B	numeric	numeric	wsml:greater-than(A,B)
A >= B	numeric	numeric	wsml:greater-equal(A,B))

Table D.2: WSML infix operators and corresponding datatype predicates

We list the built-in functions and their translation to datatype predicates in Table D.3. In the table, ?x1 represents a unique newly introduced variable, which stands for the range of the function.

Operator	Datatype (A)	Datatype (B)	Predicate
A + B	numeric	numeric	wsml:numeric-add(?x1,A,B)
A - B	numeric	numeric	wsml:numeric-subtract(?x1,A,B)
A * B	numeric	numeric	wsml:numeric-multiply(?x1,A,B)
A / B	numeric	numeric	wsml:numeric-divide(?x1,A,B)

Table D.3: Translation of WSML infix operators to datatype predicates

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

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

1. Select an atomic occurrence of a datatype function symbol. An atomic occurrence is an occurrence of the function symbol with only identifiers (which can be variables) as arguments.
2. Replace this occurrence with a newly introduced variable and append to the conjunction of which the function symbols is part the datatype predicate, which corresponds with the function symbol where the first argument (which represents the range) is the newly introduced variable.

3. If there are still occurrences of function symbols in the formula, go back to step (1), otherwise, return the formula.

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

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

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

?w = ?x1 + ?z **and** wsml:numeric-add(?x1, ?x, ?y)

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

?w = ?x2 **and** wsml:numeric-add(?x1, ?x, ?y) **and** wsml:numeric-add(?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:numeric-equal(?w, ?x2) **and** wsml:numeric-add(?x1, ?x, ?y) **and** wsml:numeric-add(?x2, ?x1, ?z)

Appendix E. A list of all WSML keywords

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

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

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

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

Keyword	Common element	Core	Flight	Rule	Full
wsmlVariant	2.2.1	+	+	+	+
namespace	2.2.2	+	+	+	+
targetNamespace	2.2.2	+	+	+	+
nonFunctionalProperties	2.2.3	+	+	+	+
endNonFunctionalProperties	2.2.3	+	+	+	+
nfp	2.2.3	+	+	+	+
endnfp	2.2.3	+	+	+	+
importedOntologies	2.2.4	+	+	+	+
usedMediators	2.2.5	+	+	+	+
ontology	2.4	+	+	+	+
goal	2.5	+	+	+	+
ooMediator	2.6	+	+	+	+
ggMediator	2.6	+	+	+	+
wgMediator	2.6	+	+	+	+

Table E.1: WSML keywords

Keyword	Common element	Core	Flight	Rule	Full
wwMediator	2.6	+	+	+	+
webService	2.7	+	+	+	+
concept		3.2.1	+	+	+
subConceptOf		3.2.1	+	+	+
ofType		3.2.1	4.2.1	+	+
impliesType		3.2.1	+	+	+
transitive		3.2.1	+	+	+
symmetric		3.2.1	+	+	+
inverseOf		3.2.1	+	+	+
reflexive			4.2.1	+	+
relation		3.2.2	+	+	+
subRelationOf		3.2.2	+	+	+
domain		3.2.2	+	+	+
range		3.2.2, 3.2.3	+	+	+
function		3.2.3	+	+	+
instance		3.2.4	+	+	+
relationInstance		3.2.4	+	+	+
memberOf		3.2.4	+	+	+
hasValue		3.2.4	+	+	+
datatype		3.2.?	+	+	+
axiom		3.2.5	+	+	+
definedBy		3.2.5	+	+	+
source	2.6	+	+	+	+
target	2.6	+	+	+	+
useService	2.6	+	+	+	+
useCapability	2.7	+	+	+	+
capability	2.7	+	+	+	+
precondition	2.7	+	+	+	+
postcondition	2.7	+	+	+	+
assumption	2.7	+	+	+	+
effect	2.7	+	+	+	+
useInterface	2.7	+	+	+	+

Keyword	Common element	Core	Flight	Rule	Full
interface	2.7	+	+	+	+
choreography	2.7	+	+	+	+
orchestration	2.7	+	+	+	+

Table E.1: WSML keywords

Appendix F. Changelog and Discussion Issues

F.1. Changelog

Compared with the previous version of this document (2004-09-26), the following changes have been made:

The major editorial changes are:

- Former authors have been removed; they are now mentioned in the acknowledgement

The changes in the content are:

- The Grammar for WSML has been added in Appendix A.1
- A full reworked WSML example has been added in Appendix A.1
- URIs are now replaced by their successor IRI (Internationalized Resource Identifiers)
- Variables may only be used inside logical expressions.
- WSML constructs are now explained using fragments of the grammar
- special non-functional properties introduced in D2 are now defined in the WSML namespace instead of special keywords in the language
- Attribute values can have non-functional properties associated with them
- The delimiters for IRIs have been changed to '<"' and '">'
- terms can not be used as identifiers for other terms or predicates; e.g. constructs such as f(a)(b) are not possible.
- The mapping of the WSML-Core conceptual syntax has been updated

Major issues not yet addressed (because of time constraints of the authors) in this version of the deliverable are:

- Update of the mapping of logical expressions to OWL
- Characterization of the allowed logical expressions in WSML-Core (closely related with the mapping of logical expressions to OWL)
- Mapping human-readable syntax to XML
- The logical expression syntax needs to be re-defined.
- The XML example need to be updated to the revised XML syntax.
- Appendix E should have a table listing all the keywords in the logical expressions
- The WSML vocabulary needs to be formally defined

Major future work:

- Update of the WSML example in the appendix as soon as the syntax is fixed and making sure that all examples in the document use this parts of this integrated example.
- Specification of the RDF syntax (once agreement is reached about the overall approach; see also discussion issues in the next section)
- Specification of WSML-DL when the language is required
- Specification of WSML-Rule once the required features of the language have become clear
- Specification of WSML-Full once the approach for defining the semantics is clear
- Specification of the semantics of all WSML variants

F.2. Discussion Issues

The following are the discussion points raised in the previous version of this document (Oct 25th), along with the outcome of the discussions. The outcome of the discussions is reflected in the current version of the document.

- Section 9.2 Representing WSML using RDF contains a number of possible directions for encoding WSML in RDF. These possible directions should be discussed. Notice that I (Jos) have suggested to **not** layer on top of RDFS (see the section for my arguments).
  Outcome of the discussion: Because of political and convenience reasons we will make a complete RDF serialization of WSML
- Basic namespace for WSML: http://www.wsmo.org/2004/wsml
  Namespaces for specific versions: http://www.wsmo.org/2004/wsml/v*x.y*
  Identifiers for WSML variants: http://www.wsmo.org/2004/wsml/wsml-*variant*
  Outcome of the discussion: We will use these namespaces
- One XML schema for all variants? Originally, the approach for the specification of the XML syntax for WSML was to create different schemas for each of the variants. However, by using the wsmlVariant keyword there is already a way to specify the desired WSML variant. Thus, it should be sufficient to have just one schema for all variants. **Outcome of the discussion:** There will be one XML schema for all variants, as suggested
- One BNF grammar for all variants? Similar to the previous discussion issue. **Outcome of the discussion:** There will be one grammar for all variants, as suggested
- The examples for the postcondition in the goal specification (Section 2.4.1) was taken from d3.3. However, the meaning of the logical expression is not really clear. Should this formula really be in WSML-Full and is it correct that none of the variables is quantified? **Outcome of the discussion:** We will wait for the discussion on the modeling of Web Services

The following discussion issues have arisen for this version of the document:

- Which features are required from WSML-Rule? Consideration of features are: function symbols, HiLog support, Transaction Logic support, disjunction in the head, classical negation.
- The keyword **definedBy** in concept definitions might be misleading, because this logical expression usually only contains a part of the definition of the concept (i.e. it does not include attributes and subconcept assertions). Furthermore, there might also be constraints which are related with the concept. A proposal by Axel is to introduce the keyword **associatedAxioms** to refer to axioms which are related to the concept.
- Do we need special syntax for evaluation of built-in functions vs. function symbols as term constructors?
- Should we drop the target namespace? It is possible to use any identifier in the specification of a certain element and thus it seems superfluous and actually confusing to allow a target namespace.
- Should we drop named parameters? Most of the things you want to do with named parameters, you can do with concepts and with concepts you're much more flexible. Furthermore, it is highly confusing for the user to have both concepts with attributed and relations with parameters.
- Making **wsmlVariant** optional and default to wsml-full? If this is the case, the client can just work with that part of the specification he can work with.

Footnotes

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

[2]The complexity of query answering for a langauge 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.