D16.1v0.21 The Web Service Modeling Language WSML

WSML Final Draft 5 October 2005

Final version
http://www.wsmo.org/TR/d16/d16.1/v0.21/20051005/

Latest version
http://www.wsmo.org/TR/d16/d16.1/v0.21/

Previous version
http://www.wsmo.org/TR/d16/d16.1/v0.21/20050923/

Editor:
Jos de Bruijn

Authors:
Jos de Bruijn
Holger Lausen
Reto Krummenacher
Axel Polleres
Livia Predoi
Michael Kifer
Dieter Fensel

Reviewers:
Ian Horrocks
Jeff Pan

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

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

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

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

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

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

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

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

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

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

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

All tools and resources related to WSML can be found at: [http://www.wsmo.org/wsml/wsml–syntax](http://www.wsmo.org/wsml/wsml–syntax)

This version (v0.21) of D16.1 constitutes a minor revision of version v0.2. A number of errors has been corrected and a number of omissions has been rectified. For the full list of changes in this version, please refer to the **Appendix F: Changelog**.
Table of Contents

PART I: PRELUDE

1. Introduction
   • 1.1. Structure of the deliverable

PART II: WSML VARIANTS

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

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

4 WSML−DL
   • 4.1. WSML−DL Logical Expression Syntax

5 WSML−Flight
   • 5.1 WSML−Flight Syntax Basics
   • 5.2. WSML−Flight Ontologies
   • 5.3. Goals in WSML−Flight
   • 5.4. Mediators in WSML−Flight
   • 5.5. Web Services in WSML−Flight
   • 5.6. WSML−Flight Logical Expression Syntax
   • 5.7. Differences between WSML−Core and WSML−Flight

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

7 WSML−Full
7.1. Differences between WSML−DL and WSML−Full
7.2. Differences between WSML−Rule and WSML−Full

8. WSML Semantics
8.1. Mapping Conceptual Syntax to Logical Expression Syntax
8.2. Preprocessing steps
8.3. WSML−Core Semantics
8.4. WSML−Flight Semantics
8.5. WSML−Rule Semantics
8.6. WSML−DL Semantics

PART III: THE WSML EXCHANGE SYNTAXES
9 XML Syntax for WSML
10 RDF Syntax for WSML
11 Mapping to OWL
11.1. Mapping WSML−Core to OWL DL
11.2. Mapping OWL DL to WSML−Core

PART IV: FINALE
12. Implementation Efforts

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

Appendix B. Schemas for the XML Exchange Syntax

Appendix C. Built−ins in WSML
C.1. WSML Datatypes
C.2. WSML Built−in Predicates
C.3. Translating Built−in Symbols to Predicates

Appendix D. WSML Keywords

Appendix E. Relation to WSMO Conceptual Model

Appendix F. Changelog

References
Acknowledgements
PART I: PRELUDE

1. Introduction

The Web Service Modeling Ontology WSMO [Roman et al. 2005] proposes a conceptual model for the description of Ontologies, Semantic Web services, Goals, and Mediators, providing the conceptual grounding for Ontology and Web service descriptions. In this document we take the conceptual model of WSMO as a starting point for the specification of a family of Web Service description and Ontology specification languages. The Web Service Modeling Language (WSML) aims at providing means to formally describe all the elements defined in WSMO. The different variants of WSML correspond with different levels of logical expressiveness and the use of different languages paradigms. More specifically, we take Description Logics, First–Order Logic and Logic Programming as starting points for the development of the WSML language variants. The WSML language variants are both syntactically and semantically layered. All WSML variants are specified in terms of a human–readable syntax with keywords similar to the elements of the WSMO conceptual model. Furthermore, we provide XML and RDF [RDF] exchange syntaxes, as well as a mapping between WSML ontologies and OWL ontologies for interoperability with OWL–based applications.

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

1.1. Structure of the Document

The remainder of this document is structured as follows:

PART II: WSML VARIANTS

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

PART III: THE WSML EXCHANGE SYNTAXES

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

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

**Appendix Guide**

This document contains a number of appendices:

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

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

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

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

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

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

WSML–Full
WSML–Full unifies WSML–DL and WSML–Rule under a First–Order umbrella with extensions to support the nonmonotonic negation of WSML–Rule. It is yet to be investigated which kind of formalisms are required to achieve this. Possible formalisms are Default Logic, Circumscription and Autoepistemic Logic.
As can be seen from Figure 2, WSML has two alternative layerings, namely, WSML-Core → WSML-DL → WSML-Full and WSML-Core → WSML-Flight → WSML-Rule → WSML-Full. In both layerings, WSML-Core is the least expressive and WSML-Full is the most expressive language. The two layerings are to a certain extent disjoint in the sense that interoperation between the Description Logic variant (WSML-DL) on the one hand and the Logic Programming variants (WSML-Flight and WSML-Rule) on the other, is only possible through a common core (WSML-Core) or through a very expressive (undecidable) superset (WSML-Full). However, there are proposals which allow interoperation between the two while retaining decidability of the satisfiability problem, either by reducing the expressiveness of one of the two paradigms, thereby effectively adding the expressiveness of one of the two paradigms to the intersection (cf. [Levy & Rousset, 1998]) or by reducing the interface between the two paradigms and reason with both paradigms independently (cf. [Elter et al., 2004]).

The only languages currently specified in this document are WSML-Core, WSML-Flight and WSML-Rule. WSML-DL will correspond (semantically) with the Description Logic $SHIQ(D_n)$, extended with more extensive datatype support.

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

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

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

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

A WSML document has the following structure:

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

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

2.1 WSML Syntax Basics

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

Nonetheless, attribute names are global and it is possible to specify global behavior of attributes through logical expressions. However, we do not expect this to be necessary in the general case and we strongly advise against it. In case one needs to model a property which is independent of the concept definition, this property is most likely a relation rather than an attribute and thus should be modeled as a relation.

It is often possible to specify a list of arguments, for example for attributes. Such argument lists in WSML are separated by commas and surrounded by curly brackets. Statements in WSML start with a keyword and can be spread over multiple lines.

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

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

2.1.1 Namespaces in WSML

Namespaces were first introduced in XML [XML–NAMESPACES–1.1] for the purpose of qualifying names which originate from different XML documents. In XML, each qualified name consists of a tuple <namespace, localname>. RDF [RDF] adopts the mechanism of namespaces from XML with the difference that qualified names are not treated as tuples, but rather as abbreviations for full URIs.
WSML adopts the namespace mechanism of RDF. A namespace can be seen as part of an IRI (see the next Section). The WSML keywords themselves belong to the the namespace http://www.wsmo.org/wsmi/wsml-syntax# (commonly abbreviated as 'wsml').

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

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

2.1.2 Identifiers in WSML

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

Data values

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

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

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

Examples of data values:

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

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

```plaintext
age ofType _integer
locationOfType _iri
hasChildrenOfType _boolean
```

WSML allows the following syntactical shortcuts for particular datatypes:

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

```plaintext
integer = ['-']? _digit+
```
decimal  =  `\-`?  digit+  `.`?  digit+
string   =  `"`?  string_content  `"`?  `"`?  not_escape_char  `"`
string_content = escaped_char | not_escape_char not_dquote

Appendix C lists the built-in predicates which any conforming WSML application must be able to support, as well as the infix notation which serves as a shortcut for the built-ins.

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

Internationalized Resource Identifiers

The IRI (Internationalized Resource Identifier) [Duerst & Suignard, 2005] mechanism provides a way to identify resources. IRIs may point to resources on the Web (in which case the IRI can start with 'http://'), but this is not necessary (e.g., books can be identified through IRIs starting with 'urn:isbn:'). The IRI proposed standard is a successor to the popular URI standard. In fact, every URI is an IRI.

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

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

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

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

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

Examples of full IRIs in WSML:

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

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

- dc#title (http://purl.org/dc/elements/1.1#title)
- foaf#name (http://xmlns.com/foaf/0.1/name)
- xsd#string (http://www.w3.org/2001/XMLSchema#string)
- Person (http://example.org/#Person)
- hasChild (http://example.org/#hasChild)
WSML defines the following two IRIs: http://www.wsmo.org/wsml/wsml-syntax#true and http://www.wsmo.org/wsml/wsml-syntax#false, which stand for universal truth and universal falsehood, respectively. Note that for convenience we typically use the abbreviated sQName form (where wsml stands for the default WSML namespace http://www.wsmo.org/wsml/wsml-syntax#): wsml#true, wsml#false. Additionally, WSML allows the keywords ‘true’ and ‘false’ in the human-readable syntax.

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

Anonymous Identifiers

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

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

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

```plaintext
anonymous  = ‘_#’
nb_anonymous = ‘_#’ digit+
```

Take as an example the following logical expressions:

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

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

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

- In case the identifier of an ontology, web service, goal or mediator is omitted or denoted by an anonymous ID, the identifier is assumed to be the same as the locator of the specification, i.e., the location where the specification was found.
- In case the identifier of a WSML element (e.g., concept, relation, postcondition) has been omitted, the unnumbered anonymous identifier ‘_#’ is used to identify the element.

```plaintext
id  =  iri
    | anonymous
    | ‘true’
    | ‘false’
idlist = id
    | ‘{ id ( , id )* }’
```
If the same identifier is used for different definitions, it is interpreted differently, depending on the context. In a concept definition, an identifier is interpreted as a concept; in a relation definition this same identifier is interpreted as a relation. If, however, the same identifier is used in separate definitions, but with the same context, then the interpretation of the identifier has to conform to both definitions and thus the definitions are interpreted conjunctively. For example, if there are two concept definitions which are concerned with the same concept identifier, the resulting concept definition includes all attributes of the original definitions and if the same attribute is defined in both definitions, the range of the resulting attribute will be equivalent to the conjunction of the original attributes.

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

Invalid identifiers

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

2.1.3 Comments in WSML

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

\[
\begin{align*}
\text{comment} & = \text{short_comment} \mid \text{long_comment} \\
\text{short_comment} & = ('/'' | 'comment ') \text{ not_cr_if_eol} \\
\text{long_comment} & = '/*' \text{ long_comment_content} '*/'
\end{align*}
\]

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

Examples:

```plaintext
/* illustrating a multi line
* comment
*/
// a one-line comment
comment another one-line comment
```

2.1.4 WSML MIME type

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

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

```
application/x–wsml
```

WSML/XML documents should have the MIME type: application/x–wsml+xml

WSML/RDF documents in the XML serialization of RDF should have the usual MIME type: application/rdf+xml

2.2 WSML Elements

This section describes the elements in common between all types of WSML specifications and all WSML variants. The elements described in this section are used in Ontology, Goal, Mediator and Web service specifications. The elements specific to a type of specification are described in subsequent sections. Because all elements in this section are concerned with meta–information about the specification and thus do not depend on the logical formalism underlying the language, these elements are shared among all WSML variants.
In this section we only describe how each element should be used. The subsequent sections will describe how these elements fit in the specific WSML descriptions.

### 2.2.1 WSML Variant

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

<table>
<thead>
<tr>
<th>WSML Variant</th>
<th>IRI</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSML–Core</td>
<td><a href="http://www.wsmo.org/wsml%E2%80%93syntax/wsml-core">http://www.wsmo.org/wsml–syntax/wsml-core</a></td>
</tr>
</tbody>
</table>

**Table 2.1: WSML variant identifiers**

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

**wsmlVariant** = ‘wsmlVariant’ full iri

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

```wsmo
wsmlVariant _http://www.wsmo.org/wsml/wsml-syntax/wsml-flight_
```

When the intended WSML variant is explicitly stated, tools can immediately recognize the intention of the author and return an exception if the specification does not conform to the syntactical restrictions imposed by the intended variant. This generally helps developers of WSML specifications to stay within desired limits of complexity and to communicate their desires to others.

### 2.2.2 Namespace References

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

```wsmo
namespace = ‘namespace’ prefixdefinitionlist
prefixdefinitionlist = full iri
| ;( prefixdefinition ( ; prefixdefinition )*)
prefixdefinition = name full iri
| full iri
```

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

```wsmo
namespace _http://www.example.org/ontologies/example#_,
dc _http://purl.org/dc/elements/1.1#_,
foaf _http://xmlns.com/foaf/0.1#_,
wsml _http://www.wsmo.org/wsml–syntax#_,
loc _http://www.wsmo.org/ontologies/location#_,
cco _http://example.org/cdoMediator#_
```

```wsmo
namespace _http://www.example.org/ontologies/example#_
```

### 2.2.3 Header

Any WSML specification may have non–functional properties, may import ontologies and may use
Non–Functional Properties

Non–functional properties may be used for the WSML document as a whole but also for each element in the specification. Non–functional property blocks are delimited with the keywords nonFunctionalProperties and endNonFunctionalProperties or the short forms nfp and endnfp. Following the keyword is a list of attribute values, which consists of the attribute identifier, the keyword hasValue and the value for the attribute, which may be any identifier and can thus be an IRI, a data value, an anonymous identifier or a comma–separated list of the former, delimited with curly brackets. The recommended properties are the properties of the Dublin Core [Weibel et al. 1998], but the list of properties is extensible and thus the user can choose to use properties coming from different sources. WSMO [Roman et al., 2005] defines a number of properties which are not in the Dublin Core. These properties can be used in a WSML specification by referring to the WSML namespace (http://www.wsmo.org/wsml/wsml-syntax#). These properties are: wsml#version, wsml#accuracy, wsml#financial, wsml#networkRelatedQoS, wsml#performance, wsml#reliability, wsml#robustness, wsml#scalability, wsml#security, wsml#transactional, wsml#trust (here we assume that the prefix wsml has been defined as referring to the WSML namespace; see Section 2.1.1). For recommended usage of these properties see [Roman et al., 2005]. Following the WSML convention, if a property has multiple values, these are separated by commas and the list of values is delimited by curly brackets.

```
nfp = ‘nfp’ attributevalue’ ‘endnfp’
     | ‘nonFunctionalProperties’ attributevalue’ ‘endNonFunctionalProperties’
```

Example:
```
nonFunctionalProperties
dctitle hasValue “WSML example ontology”
dcsubject hasValue “family”
dcdescription hasValue “fragments of a family ontology to provide WSML examples”
dccreator hasValue “...”
dctitle hasValue “...”
dcsubject hasValue “...”
dcdescription hasValue “...”
dccreator hasValue “...”
dctitle hasValue “...”
dcsubject hasValue “...”
dcdescription hasValue “...”
dccreator hasValue “...”
endnonFunctionalProperties
```

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

Importing Ontologies

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

```
importsontology = ‘importsOntology’ idlist
```

Example:
```
importsOntology ("...", "...")
```

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

Using Mediators

Mediators are used to link different WSML elements (ontologies, goal, web services) and resolve heterogeneity between the elements. Mediators are described in more detail in Section 2.5. Here, we are only concerned with how mediators can be referenced from a WSML specification. Mediators are currently underspecified and thus this reference to the use of mediators can be seen as a placeholder.

The (optional) used mediators block is identified by the keywords usesMediator which is followed by one or more identifiers of WSML mediators. The types of mediators which can be used are constrained by the type of specification. An ontology allows for the use of different mediators than, for example, a goal or a web service. More details on the use of different mediators can be found in Section 2.5. The type of the mediator is reflected in the mediator specification itself and not in the reference to the mediator.

usesmediator = ‘usesMediator’ idlist

Example:

usesMediator _,”http://example.org/ooMediator”

2.3 Ontology Specification in WSML

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

Example:

ontology family

An ontology specification document in WSML consists of:

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

In this section we explain the ontology modeling elements in the WSML language. The modeling elements are based on the WSMO conceptual model of ontologies [Roman et al., 2005].

2.3.1 Concepts

A concept definition starts with the concept keyword, which is optionally followed by the identifier of the concept. This is optionally followed by a superconcept definition which consists of the keyword subConceptOf followed by one or more concept identifiers (as usual, if there is more than one, the list is comma–separated and delimited by curly brackets). This is followed by an optional nonFunctionalProperties block and zero or more attribute definitions.

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

concept = ‘concept’ id  superconcept?  nfp?  attribute*
superconcept = ‘subConceptOf’ idlist

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

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

WSML supports defined concepts through the use of axioms (see Section 2.3.4). The logical expression contained in the axiom should reflect an equivalence relation between a class membership expression on one side and a conjunction of class membership expressions on the other side, each with the same variable. Thus, such a definition should be of the form:

\[ \exists x \text{memberOf } A \equiv \exists x \text{memberOf } B_1 \text{ and } \ldots \text{ and } \exists x \text{memberOf } B_n \]

With \( A \) and \( B_1, \ldots, B_n \) 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:

\[ \text{axiom humanDefinition} \]
\[ \text{definedBy} \]
\[ \exists x \text{memberOf } \text{Human} \equiv \exists x \text{memberOf } \text{Primate} \text{ and } \exists x \text{memberOf } \text{LegalAgent}. \]

**Attributes**

Two important features of attribute modeling which set WSML apart from OWL are cardinality constraints and attribute range constraints. OWL offers cardinality and range restrictions on attributes which serve to create additional (monotonic) inferences such as existence or equality of objects or membership in a certain class. For many users these restrictions show unintuitive behavior from the viewpoint of classical rule languages and databases [de Bruin et al., 2005]: Means to actually check the data in your knowledge base with respect to integrity constraints are missing in OWL. WSML allows the specification of cardinality and range constraints which are defined like integrity constraints in databases, i.e., in the case of violation of a constraint, a given ontology is inconsistent. This additional feature allows to check the integrity of a closed set of data, implicitly the modeler to express a local a closed world view on her/his published data.

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

An attribute definition of the form \( A \text{ ofType } D \), where \( A \) is an attribute identifier and \( D \) is a concept identifier, is a constraint on the values for attribute \( A \). If the value for the attribute \( A \) is not known to be of type \( D \), the constraint is violated and the attribute value is inconsistent with respect to the ontology. This notion of constraints corresponds with the usual database–style constraints.
The keyword **impliesType** can be used for inferring the type of a particular attribute value. An attribute definition of the form \( A \text{ impliesType} D \), where \( A \) is an attribute identifier and \( D \) is a concept identifier, implies membership of the concept \( D \) for all values of the attribute \( A \). Please note that if the range of the attribute is a datatype, the semantics of \( \text{ofType} \) and \( \text{impliesType} \) coincide, because datatypes have a known domain and thus values cannot be inferred to be of a certain datatype.

Attributes which have a datatype as a range can be distinguished from regular attributes through the meta--concept \_datatype. Each datatype used in WSML is a member of this meta--concept.

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

The cardinality constraints for a single attribute are specified by including two numbers between parentheses \( (\ ) \), indicating the minimal and maximal cardinality, after the ofType (or impliesType) keyword. The first number indicates the minimal cardinality. The second number indicates the maximal cardinality, where \( ^* \) stands for unlimited maximal cardinality (and is not allowed for minimal cardinality). It is possible to write down just one number instead of two, which is interpreted as both a minimal and a maximal cardinality constraint. When the cardinality is omitted, then it is assumed that there are no constraints on the cardinality, which is equivalent to \( (0 ^*) \). Note that a maximal cardinality of 1 makes an attribute functional.

\[
\begin{align*}
\text{attribute} & = [\text{attr}]: \text{id} \ _\text{attributefeature}^* \ _\text{att_type} \ _\text{cardinality}? \ [\text{type}]: \text{idlist} \ _\text{nfp} \\
\text{att_type} & = \text{ofType} \\
\text{cardinality} & = (\_\text{min_cardinality}; \ _\text{max_cardinality})? \\
\text{cardinality_number} & = \text{finite_cardinality} \ _\text{digit}^+ \ _\text{infinite_cardinality}^* \\
\text{attributefeature} & = \text{transitive} \\
\text{symmetric} & \mid \text{inverseOf} (\_\text{id} ^*) \\
\text{reflexive} & \\
\end{align*}
\]

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 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 \( a \).

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

```xml
calendar Human
  nonFunctionalProperties
dcl description hasValue "concept of a human being"
endNonFunctionalProperties
hasName ofType foaf:name
hasParent inverseOf(hasChild) impliesType Human
hasChild impliesType Human
hasAncestor transitive impliesType Human
hasWeight ofType (1) _float
hasHeight ofType (1) _float
hasbirthdate ofType (1) _date
hasObit ofType (0 1) _date
hasBirthplace ofType (1) _location
isMarriedTo symmetric impliesType (0 1) Human
hasCitizenship ofType oco:country

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

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

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

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

relation distance (ofType City, ofType City, impliesType _decimal) subRelationOf measurement
relation distance/3

As for concepts, the exact meaning of a relation can be defined using axioms. For example one could axiomatize the transitive closure for a property or further restrict the domain of one of the parameters. As with concepts, it is recommended that related axioms are indicated using the non–functional property dclRelation.

2.3.3 Instances

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

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

Example:

instance Mary memberOf (Parent, Woman)
  nfp
dclDescription hasValue "Mary is parent of the twins Paul and Susan"
endnfp
hasName hasValue "Mary Smith"
hasBirthdate hasValue date(1949,9,12)
hasChild hasValue (Paul, Susan)

Instances explicitly specified in an ontology are those which are shared together as part of the ontology. However, most instance data exists outside the ontology in private data stores. Access to these instances, as described in [Roman et al., 2005], is achieved by providing a link to an instance store. Instance stores contain large numbers of instances and they are linked to the ontology. We do not restrict the user in the way an instance store is linked to a WSML ontology. This would be done outside the ontology definition, since an ontology is shared and can thus be used in combination with different instance stores.

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.

\[
\text{relationInstance} = \text{relationInstance}[\text{name}: \text{id} [\text{relation}: \text{id} (' value (' value)*')] nfp?]
\]

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

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

### 2.3.4 Axioms

An axiom definition starts with the axiom keyword, followed by the name (identifier) of the axiom. This is followed by an optional nonFunctionalProperties block and a logical expression preceded by the definedBy keyword. The logical expression must be followed by either a blank or a new line. The language allowed for the logical expression is explained in Section 2.6.

\[
\text{axiom} = \text{axiom} \text{ axiomdefinition} \\
\text{axiomdefinition} = \text{id} \\
| \text{id} nfp? \log \text{ definition} \\
\log \text{ definition} = \text{definedBy} \log \text{ expr+}
\]

Example of a defining axiom:

\[
\text{axiom humanDefinition} \\
\text{definedBy} \\
?x \text{ memberOf Human equivalent} \\
?x \text{ memberOf Animal and} \\
?x \text{ memberOf LegalAgent.}
\]

WSML allows the specification of database–style constraints. Below is an example of a constraining axiom:

\[
\text{axiom humanBMIConstraint} \\
\text{definedBy} \\
| \text{bodyMassIndex(bmi hasValue ?b, length hasValue ?l, weight hasValue ?w)} \\
| \text{and ?x memberOf Human and} \\
| \text{length hasValue ?l, weight hasValue ?w, bmi hasValue ?b).}
\]

### 2.4 Capability and Interface Specification in WSML

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

#### 2.4.1 Capabilities

A capability constitutes a formal description of the functionality requested from or provided by a web service. The preconditions describe conditions on the input of the service, the postconditions describe the relation between the input and the output of the service, the assumptions describe what must hold (but cannot be checked beforehand) of the state of the world for the web service to be able to execute successfully, and the effects describe real–world effects of the execution of the web service which are not reflected in the output.

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

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

```plaintext
capability = 'capability' id? header? sharedvardef? pre post ass or eff
sharedvardef = 'sharedVariables' variablelist
pre post ass or eff = 'precondition' axiomdefinition
| 'postcondition' axiomdefinition
| 'assumption' axiomdefinition
| 'effect' axiomdefinition
```

Below is an example of a capability specified in WSML:

```plaintext
capability
  sharedVariables ?child
  precondition
    nonFunctionalProperties
      dcl:description hasValue "The input has to be boy or a girl
      with birthdate in the past and be born in Germany."
    endNonFunctionalProperties
  definedBy
    ?child memberOf Child
    and (?child[hasBirthdate hasValue ?birthdate])
    and (?child[hasBirthplace hasValue ?location])
    and (?location[locatedIn hasValue ?location])
    or (?child[hasParent hasValue /parent] and
    ?parent[hasCitizenship hasValue ?citizenId])
  .

effect
  nonFunctionalProperties
    dcl:description hasValue "After the registration the child
    is a German citizen"
  endNonFunctionalProperties
  definedBy
    ?child memberOf Child
    and (?child[hasCitizenship hasValue ?citizenId]).
```

### 2.4.2 Interface

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

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

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

Below is an example of an interface and an example of references to multiple interfaces:

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

interface ("http://example.org/mychoreography","http://example.org/myorchestration")
```

We do not define ways to specify the choreography and orchestration here. Instead, we refer the reader to the corresponding WSMO deliverable D14 [Scicluna et al., 2009].
2.5 Goal Specification in WSML

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

Example:

```xml
<goal _="http://example.org/Germany/GetCitizenShip"/>
```

A goal specification document in WSML consists of:

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

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

2.6 Mediator Specification in WSML

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

```
mediator = ooMediator
          | ggMediator
          | wgMediator
          | wwMediator
```

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

Example:

```xml
<ooMediator _="http://example.org/ooMediator"/>
```

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

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

target = 'target' id
```

2.6.1 ooMediators

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

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

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

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

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

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

### 2.6.2 wwMediators

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

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

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

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

### 2.6.3 ggMediators

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

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

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

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

### 2.6.4 wgMediators

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

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

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

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

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

None of the elements in a mediator has any meaning in the logical language. In fact, the complexity of a mediator is hidden in the actual description of the mediator. Instead, the complexity is either in the implementation of the mediation service, in which case WSML does not support the description because WSML is only concerned with the interface description, or in the functional description of the web service or the goal which is used to specify the desired mediation service. As discussed in [Keller et al., 2005], these descriptions often need a very expressive language.

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

A web service specification document in WSML consists of:

```xml
webService = 'webService' id? header* capability? interface*
```

Example:

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

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

### 2.8 Logical Expressions in WSML

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

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

**Variables**

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

The scope of a variable is always defined by its quantification. If a variable is not quantified inside a formula, the variable is implicitly universally quantified outside the formula, unless the formula is part of a capability description and the variable is explicitly mentioned in the **sharedVariables** block.

```xml
variable = '?' alphanum*
```

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

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

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

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

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

WSML allows the following logical connectives: and, or, implies, impliedBy, equivalent, neg, naf, forall and exists and the following auxiliary symbols: ('', ',)', '[', ']'), '.', '=', '!', ':=', ':=', memberOf, hasValue, subConceptOf, oType, and impliesType. Furthermore, WSML allows use of the symbol ':-' for Logic Programming rules and the use of the symbol '!'- for database–style constraints.
Definition 2.2 defines the set of terms $\text{Term}(V)$ for a given vocabulary $V$.

Definition 2.2. Given a vocabulary $V$, the set of terms $\text{Term}(V)$ in WSML is defined as follows:

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

As usual, the set of ground terms $\text{GroundTerm}(V)$ is the maximal subset of $\text{Term}(V)$ which does not contain variables.

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

Definition 2.3. Given a set of terms $\text{Term}(V)$, the set of atomic formulae in $\mathcal{L}(V)$ is defined by:

- If $\alpha, \beta, \gamma \in \text{Term}(V)$ and $\gamma$ is of the form $\{ \gamma_1, \ldots, \gamma_n \}$, with $\gamma_1, \ldots, \gamma_n \in \text{Term}(V)$, then:
  o $\alpha \text{ subConceptOf } \gamma$ is an atomic formula in $\mathcal{L}(V)$. Here, $\alpha$ and $\gamma$ both identify concepts.
  o $\alpha \text{ memberOf } \gamma$ is an atomic formula in $\mathcal{L}(V)$. Here, $\alpha$ identifies an instance and $\gamma$ identifies an instance of $\alpha$.
  o $\alpha \text{ ofType } \gamma$ is an atomic formula in $\mathcal{L}(V)$. Here, $\alpha$ identifies an instance, $\beta$ identifies an attribute, and $\gamma$ identifies a concept.
  o $\alpha \text{ impliesType } \gamma$ is an atomic formula in $\mathcal{L}(V)$. Here, $\gamma$ identifies an instance, $\beta$ identifies an attribute, and $\gamma$ identifies a concept.
  o $\alpha \text{ hasValue } \gamma$ is an atomic formula in $\mathcal{L}(V)$. Here, $\alpha$ identifies an instance, $\beta$ identifies an attribute, and $\gamma$ identifies an attribute $\alpha$. These atomic formulae are also called molecules. Notice that an identifier can play a different role (instance, attribute, concept), depending on the use in a molecule.
- If $f \in V_R$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.
- If $\alpha, \beta \in \text{Term}(V)$ then $\alpha \models \beta$, $\alpha \models \beta$, $\alpha \models \beta$, $\alpha \models \beta$, $\alpha \models \beta$ are atomic formulae in $\mathcal{L}(V)$.

Given the atomic formulae, we recursively define the set of formulae in $\mathcal{L}(V)$ in definition 2.4.

Definition 2.4. The set of formulae in $\mathcal{L}(V)$ is defined by:

- Every atomic formula in $\mathcal{L}(V)$ is a formula in $\mathcal{L}(V)$.
- Let $\alpha, \beta$ be formulae which do not contain the symbols ‘$-$’ and ‘$!$’, and let $?x_1, \ldots, ?x_n$ be variables, then:
  o $\alpha \text{ and } \beta$ is a formula in $\mathcal{L}(V)$.
  o $\alpha \text{ or } \beta$ is a formula in $\mathcal{L}(V)$.
  o $\text{neg } \alpha$ is a formula in $\mathcal{L}(V)$.
  o $\text{naf } \alpha$ is a formula in $\mathcal{L}(V)$.
  o $\text{forall } ?x_1, \ldots, ?x_n (\alpha)$ is a formula in $\mathcal{L}(V)$.
  o $\text{exists } ?x_1, \ldots, ?x_n (\alpha)$ is a formula in $\mathcal{L}(V)$.
  o $\alpha \text{ implies } \beta$ is a formula in $\mathcal{L}(V)$.
  o $\alpha \text{ impliedBy } \beta$ is a formula in $\mathcal{L}(V)$.
  o $\alpha \text{ equivalent } \beta$ is a formula in $\mathcal{L}(V)$.
  o $\alpha \models \beta$ is a formula in $\mathcal{L}(V)$. This formula is called an $LP$ (Logic Programming) rule. $\alpha$ is called the head and $\beta$ is called the body of the rule.
  o $\text{!}$ $\alpha$ is a formula in $\mathcal{L}(V)$. This formula is called a constraint. We say $\alpha$ is a constraint of the knowledge base.

Note that WSML allows the symbols $\text{->}$, $\text{<}$ and $\text{<->}$ as synonyms for implies, impliedBy, and equivalent, respectively.

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

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

\[
\text{Human} \; \text{subConceptOf} \; \text{Mammal} \\
\quad \text{and} \; \text{Human}[\text{hasName ofType foaf#name}] \; \text{and} \; \text{Human}[\text{hasChild impliesType Human}]
\]

can be written as

\[
\text{Human}[\text{hasName ofType foaf#name}, \text{hasChild impliesType Human}] \; \text{subConceptOf} \; \text{Mammal}
\]

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

No human can be both male and female:

\[
\neg \exists x [\text{gender hasValue } (?y, ?z)] \; \text{memberOf} \; \text{Human} \; \text{and} \; ?y = \text{Male} \; \text{and} \; ?z = \text{Female}.
\]

A human who is not a man is a woman:

\[
\exists x [\text{gender hasValue Woman}] \; \text{impliedBy} \; \neg \exists x [\text{gender hasValue Man}].
\]

The brother of a parent is an uncle:

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

Do not trust strangers:

\[
\exists x [\text{distrust hasValue } ?y] \; \Rightarrow \; \text{naf} \; \exists x [\text{knows hasValue } ?y].
\]
3 WSML–Core

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

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

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

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

3.1 WSML-Core Syntax Basics

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

WSML–Core inherits the namespace mechanism of WSML.

WSML–Core restricts the use of identifiers. The vocabulary of WSML–Core is separated similarly to OWL DL.

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

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

3.2. WSML-Core Ontologies

In this section we explain the restrictions on the WSML ontology modeling elements imposed by WSML–Core. The restrictions posed on the conceptual syntax for ontologies is necessary because of the restriction imposed on WSML–Core by the chosen underlying logical formalism (the intersection of Datalog and Description Logics), cf. [Grosof et al., 2003].

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

3.2.1 Concepts

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

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

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

\[
\text{attribute} = [\text{attr}]: \text{id, att_type; \text{type: id, nfp}]
\]

### 3.2.2 Relations

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

\[
\text{relation} = \text{'relation id '2? paramtyping? superrelation? nfp?}
\]

\[
\text{paramtyping} = \text{'( 'impliesType idlist 'idlist 'att_type idlist )'}
\]

\[
\text{superrelation} = \text{'subRelationOf idlist}
\]

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

### 3.2.3 Instances

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

### 3.2.4 Axioms

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

### 3.3. Goals in WSML–Core

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

### 3.4. Mediators in WSML–Core

Mediators in WSML–Core follow the common WSML syntax.

### 3.5. Web Services in WSML–Core

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

### 3.6. WSML–Core Logical Expression Syntax

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

Let \( V \) be a WSML–Core vocabulary. Let further \( \gamma \in V_C, \Gamma \) be either an identifier in \( V_C \) or a list of identifiers in \( V_C, \Delta \) be either an identifier in \( V_D \) or a list of identifiers in \( V_D, \Phi \in V_I, \Psi \) be either an identifier in \( V_I \) or a list of identifiers in \( V_I, p,q \in V_{RA}, s,t \in V_{RC} \), and \( Val \) be either a data value or a list of data values.
Definition 3.2. The set of atomic formulae in \( L(V) \) is defined as follows:

- \( \gamma \text{ subConceptOf } \Gamma \) is an atomic formula in \( L(V) \)
- \( \phi \text{ memberOf } \Gamma \) is an atomic formula in \( L(V) \)
- \( \{ p \text{ ofType } \Delta \} \) is an atomic formula in \( L(V) \)
- \( \{ s \text{ impliesType } \Delta \} \) is an atomic formula in \( L(V) \)
- \( \{ s \text{ impliesType } \Gamma \} \) is an atomic formula in \( L(V) \)
- \( \{ p \text{ hasValue } \psi \} \) is an atomic formula in \( L(V) \)
- \( \{ s \text{ hasValue } \psi \} \) is an atomic formula in \( L(V) \)

Let \( Var_1, Var_2, \ldots \) be arbitrary WSML variables. We call molecules of the form \( Var_i \text{ memberOf } \Gamma \) \( a\)-molecules, and molecules of the forms, \( Var_i \ p \text{ hasValue } Var_k \) and \( Var_i \ p \text{ hasValue } \{ Var_k, Var_k \} \) \( b\)-molecules, respectively.

In the following, \( F \) stands for an Lhs–formula (i.e., a formula allowed in the antecedent, or left–hand side, of an implication), with the set of Lhs–formulae defined as follows:

- Any \( b\)-molecule is an Lhs–formula
- If \( F_1 \) and \( F_2 \) are Lhs–formulae, then \( F_1 \text{ and } F_2 \) is an Lhs–formula
- If \( F_1 \) and \( F_2 \) are Lhs–formulae, then \( F_1 \text{ or } F_2 \) is an Lhs–formula

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

- Any \( a\)-molecule is an rhs–formula
- If \( G \) and \( H \) are rhs–formulae, then \( G \text{ and } H \) is an rhs–formula

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

- Any atomic formula is a formula in \( L(V) \).
- If \( F_1, \ldots, F_n \) are atomic formulae, then \( F_1 \text{ and } \ldots \text{ and } F_n \) is a formula in \( L(V) \).
- \( Var_1 \ p \text{ hasValue } Var_2 \) \( \text{ impliedByVar} \ Var_1 \ p \text{ hasValue } Var_3 \) and \( \{ p \text{ hasValue } Var_2 \} \) (globally transitive attribute/relation) is a formula in \( L(V) \).
- \( Var_1 \ p \text{ hasValue } Var_2 \) \( \text{ impliedByVar} \ Var_2 \ p \text{ hasValue } Var_3 \) (globally symmetric attribute/relation) is a formula in \( L(V) \).
- \( Var_1 \ p \text{ hasValue } Var_2 \) \( \text{ impliedByVar} \ Var_1 \ q \text{ hasValue } Var_2 \) (globally sub–attribute/relation) is a formula in \( L(V) \).
- \( Var_1 \ p \text{ hasValue } Var_2 \) \( \text{ impliedByVar} \ Var_2 \ q \text{ hasValue } Var_1 \) (globally inverse attribute/relation) is a formula in \( L(V) \).
- \( G \text{ equivalent } H \) is a formula in \( L(V) \) if it contains only one WSML variable.
- \( H \text{ impliedBy } F \) (and \( F \text{ implies } H \)) is a formula in \( L(V) \) if all the WSML variables occurring in \( H \) occur in \( F \) as well and the variable graph of \( F \) is connected and acyclic.
- Any occurrence of a molecule of the form \( Var_1 \ p \text{ hasValue } Var_2 \) in a WSML–Core clause can be interchanged with \( p(Var_1, Var_2) \) (i.e., these two forms can be used interchangeably in WSML Core).

Here, the variable graph of a logical expression \( E \) is defined as the undirected graph having all WSML variables in \( E \) as nodes and an edge between \( Var_1 \) and \( Var_2 \) for every molecule \( Var_1 \ p \text{ hasValue } Var_2 \).

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

The following are examples of WSML–Core logical expressions:

The attribute ‘hasAncestor’ is transitive:

- \( ?x \text{ hasAncestor } ?y \text{ impliedBy } ?x \text{ hasAncestor } ?z \) and \( ?y \text{ hasAncestor } ?z \).

A female person is a woman:

- \( ?x \text{ memberOf } \text{Woman} \text{ impliedBy } ?x \text{ memberOf } \text{Person and } ?x \text{ memberOf } \text{Female} \)
A student is a person:

Student subConceptOf Person.
4. WSML–DL

WSML–DL is an extension of WSML–Core to a full–fledged description logic with an expressiveness similar to OWL DL, namely SHIQ(D). WSML–DL is both syntactically and semantically completely layered on top of WSML–Core. This means that every valid WSML–Core specification is also a valid WSML–DL specification. Furthermore, all consequences inferred from a WSML–Core specification are also valid consequences of the same specification in WSML–DL. Finally, if a WSML–DL specification falls inside the WSML–Core fragment then all consequences with respect to the WSML–DL semantics also hold with respect to the WSML–Core semantics.

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

4.1. WSML–DL Logical Expression Syntax

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

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

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

- \( \phi \) memberOf \( \Gamma \) is an atomic formula in \( L(V) \)
- \( \gamma \) subConceptOf \( \Gamma \) is an atomic formula in \( L(V) \)
- \( \{ s \text{ ofType } \Delta \} \) is an atomic formula in \( L(V) \)
- \( \{ s \text{ impliesType } \Gamma \} \) is an atomic formula in \( L(V) \)
- \( \{ p \text{ impliesType } \Gamma \} \) is an atomic formula in \( L(V) \)
- \( \{ p \text{ hasValue } \psi \} \) is an atomic formula in \( L(V) \)
- \( \{ s \text{ hasValue } \varphi \} \) is an atomic formula in \( L(V) \)

Let \( \text{Var}_1, \text{Var}_2, \ldots \) be arbitrary WSML variables. We call molecules of the form \( \text{Var}_i \text{ memberOf } \Gamma \) a–molecules, and molecules of the forms \( \text{Var}_i \text{ p hasValue } \text{Var}_k \) and \( \text{Var}_i \text{ p hasValue } \{ \text{Var}_{k1}, \text{Var}_{k2} \} \) b–molecules, respectively.

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

Definition 4.2. The set of descriptions in \( L(V) \) is defined as follows:

- Any a–molecule or b–molecule is a description in \( L(V) \).
- If \( F_1 \) and \( F_2 \) are descriptions in \( L(V) \) and \( G \) is a b–molecule, then:
  - \( F_1 \text{ and } F_2 \) is a description in \( L(V) \).
  - \( F_1 \text{ or } F_2 \) is a description in \( L(V) \).
  - \( \text{neg } F_1 \) is a description in \( L(V) \).
  - \( \text{forall } \text{Var}_i \text{ G implies } F_1 \) is a description in \( L(V) \).
  - \( \text{exists } \text{Var}_i \text{ G and } F_1 \) is a description in \( L(V) \).
- \( \exists \text{Var}_1, \ldots, \text{Var}_n, G_1, \ldots, G_n, \text{ and } F_1 \text{ and neg } (\text{Var}_1 :=: \text{Var}_2) \text{ and } \ldots \text{ and neg } (\text{Var}_i :=: \text{Var}_{n+1}) \text{ and } \text{neg } (\text{Var}_{n+1} :=: \text{Var}_{n+2}) \) is a description in \( L(V) \)
- \( \text{forall } \text{Var}_1, \ldots, \text{Var}_{n+1}, G_1, \ldots, G_{n+1}, \text{ and } F_1 \text{ implies neg } (\text{Var}_1 :=: \text{Var}_2) \text{ or } \ldots \text{ or neg } (\text{Var}_i :=: \text{Var}_{n+1}) \text{ or } \ldots \text{ or neg } (\text{Var}_n :=: \text{Var}_{n+1}) \) is a description in \( L(V) \)

The variable graph of a description \( F \) is defined as the undirected graph having all WSML variables in \( F \)
as nodes and an edge between \( \operatorname{Var}_1 \) and \( \operatorname{Var}_2 \) for every molecule \( \operatorname{Var}_1[p \ \operatorname{hasValue} \ \operatorname{Var}_2]. \)

In the following, \( F_1 \), \( F_2 \) stand for WSML–DL descriptions with connected, acyclic variable graphs; furthermore, the variable graphs of \( F_1 \) and \( F_2 \) can be seen as trees which share the same root node.

**Definition 4.3.** The set of WSML–DL formulae in \( L(V) \) is defined as follows:

- Any atomic formula is a formula in \( L(V) \).
- \( F_1 \) implies \( F_2 \) is a formula in \( L(V) \).
- \( F_1 \) impliedBy \( F_2 \) is a formula in \( L(V) \).
- \( F_1 \) equivalent \( F_2 \) is a formula in \( L(V) \).
- \( \operatorname{Var}_1[p \ \operatorname{hasValue} \ \operatorname{Var}_2] \) impliedBy \( \operatorname{Var}_1[p \ \operatorname{hasValue} \ \operatorname{Var}_3] \) and \( \operatorname{Var}_2[p \ \operatorname{hasValue} \ \operatorname{Var}_4] \) (globally transitive attribute/relation) is a formula in \( L(V) \).
- \( \operatorname{Var}_1[p \ \operatorname{hasValue} \ \operatorname{Var}_2] \) impliedBy \( \operatorname{Var}_2[p \ \operatorname{hasValue} \ \operatorname{Var}_4] \) (globally symmetric attribute/relation) is a formula in \( L(V) \).
- \( \operatorname{Var}_1[p \ \operatorname{hasValue} \ \operatorname{Var}_2] \) impliedBy \( \operatorname{Var}_1[q \ \operatorname{hasValue} \ \operatorname{Var}_3] \) (globally sub-attribute/relation) is a formula in \( L(V) \).
- \( \operatorname{Var}_1[p \ \operatorname{hasValue} \ \operatorname{Var}_2] \) impliedBy \( \operatorname{Var}_2[q \ \operatorname{hasValue} \ \operatorname{Var}_3] \) (globally inverse attribute/relation) is a formula in \( L(V) \).

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

The following are examples of WSML–DL logical expressions:

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

\[
\exists x \ \operatorname{memberOf} \ \operatorname{Human} \\
\text{equivalent} \\
\exists x \ \operatorname{memberOf} \ \operatorname{Woman} \text{ or } \exists x \ \operatorname{memberOf} \ \operatorname{Man}.
\]

**The concepts Man and Woman are disjoint.**

\[
\exists x \ \operatorname{memberOf} \ \operatorname{Man} \\
\text{implies} \\
\neg \exists x \ \operatorname{memberOf} \ \operatorname{Woman}.
\]

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

\[
\exists x \ \operatorname{memberOf} \ \operatorname{Human} \\
\text{implies} \\
\exists y \ ( \\
\exists y ( \exists y ( \exists y ( \exists y \text{father hasValue } y ) \text{ and } \exists y \operatorname{memberOf} \ \operatorname{Human} ) \\
\text{ and } \\
\forall y ( \\
\exists y \exists y ( \exists y \exists y \text{father hasValue } y ) \\
\text{implies } \exists y \operatorname{memberOf} \ \operatorname{Human} ).
\]
5. WSML–Flight

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

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

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

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

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

5.1 WSML–Flight Syntax Basics

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

5.2. WSML–Flight Ontologies

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

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

5.3. Goals in WSML–Flight

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

5.4. Mediators in WSML–Flight

Mediators in WSML–Core follow the common WSML syntax.

5.5. Web Services in WSML–Flight

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

5.6. WSML–Flight Logical Expression Syntax

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

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

The syntax for logical expressions of WSML Flight is the same as described in Section 2.8 with the restrictions described in the following. We define the notion of a WSML–Flight vocabulary in Definition 5.1.

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

Definition 5.2 defines the set of WSML–Flight terms \(\text{Term}_{\text{Flight}}(V)\) for a given vocabulary \(V\).

**Definition 5.2.** Given a vocabulary \(V\), the set of terms \(\text{Term}_{\text{Flight}}(V)\) in WSML–Flight is defined as follows:

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

As usual, the set of ground terms \(\text{GroundTerm}_{\text{Flight}}(V)\) is the maximal subset of \(\text{Term}_{\text{Flight}}(V)\) which does not contain variables.

WSML–Flight does not allow the equality operator (\(\equiv\)). Therefore, the set of admissible atomic formulae in WSML–Flight does not contain \(\alpha \equiv \beta\) for terms \(\alpha, \beta\).

**Definition 5.3.** Given a set of WSML–Flight terms \(\text{Term}_{\text{Flight}}(V)\), an atomic formula in \(L(V)\) is defined by:

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

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

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

- We define the set of admissible head formulae \(\text{Head}(V)\) as follows:
  - Any atomic formula \(\alpha\) which does not contain the inequality symbol (\(!=\)) or the unification operator (\(\equiv\)) is in \(\text{Head}(V)\).
  - Let \(\alpha, \beta \in \text{Head}(V)\), then \(\alpha\) and \(\beta\) is in \(\text{Head}(V)\).
  - For \(\alpha, \beta \in \text{Head}(V)\) and \(\alpha, \beta\) do not contain \{\text{implies, impliedBy, equivalent}\}, the following formulae are in \(\text{Head}(V)\):
    - \(\alpha \text{ implies} \beta\)
    - \(\alpha \text{ impliedBy} \beta\)
    - \(\alpha \text{ equivalent} \beta\)
  - Any variable–free admissible head formula in \(\text{Head}(V)\) is a formula in \(L(V)\).
- We define the set of admissible body formulae \(\text{Body}(V)\) as follows:
  - Any atomic formula \(\alpha\) is in \(\text{Body}(V)\)
  - For any atomic formula \(\alpha, \text{naf} \alpha\) is in \(\text{Body}(V)\).
  - For \(\alpha, \beta \in \text{Body}(V)\), \(\alpha\) and \(\beta\) is in \(\text{Body}(V)\).
  - For \(\alpha, \beta \in \text{Body}(V)\), \(\alpha\) or \(\beta\) is in \(\text{Body}(V)\).
- Given a head–formula \(\beta \in \text{Head}(V)\) and a body–formula \(\alpha \in \text{Body}(V)\), \(\beta \models \alpha\) is a formula. Here we call \(\alpha\) the body and \(\beta\) the head of the formula. The formula is admissible if (1) \(\alpha\) is an admissible body formula, (2) \(\beta\) is an admissible head formula, and (3) the safety condition holds.
- Any formula of the form \(!\alpha\) with \(\alpha \in \text{Body}(V)\) is an admissible formula and is called a constraint.
Any WSML-Flight admissible formula together with a period `.` is a logical expression in WSML-Flight.

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

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

- Rules of the form $A_1$ and ... and $A_n$ :- $B$ are split into $n$ different rules:
  - $A_1$ :- $B$
  - ... 
  - $A_n$ :- $B$
- Rules of the form $A_1$ equivalent $A_2$ :- $B$ are split into 2 rules:
  - $A_1$ implies $A_2$ :- $B$
  - $A_1$ impliedBy $A_2$ :- $B$
- Rules of the form $A_1$ impliedBy $A_2$ :- $B$ are transformed to:
  - $A_1$ :- $A_2$ and $B$
- Rules of the form $A_1$ implies $A_2$ :- $B$ are transformed to:
  - $A_2$ :- $A_1$ and $B$
- Rules of the form $A$ :- $B_1$ and $(F or G)$ and $B_2$ are split into two different rules:
  - $A$ :- $B_1$ and $F$ and $B_2$
  - $A$ :- $B_1$ and $G$ and $B_2$
- Rules of the form $A$ :- $B_1$ and naf $(F and G)$ and $B_2$ are split into two different rules:
  - $A$ :- $B_1$ and naf $F$ and $B_2$
  - $A$ :- $B_1$ and naf $G$ and $B_2$
- Rules of the form $A$ :- $B_1$ and naf $(F or G)$ and $B_2$ are transformed to:
  - $A$ :- $B_1$ and naf $F$ and naf $G$ and $B_2$
- Rules of the form $A$ :- $B_1$ and naf naf $F$ and $B_2$ are transformed to:
  - $A$ :- $B_1$ and $F$ and $B_2$

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

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

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

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

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

No human can be both male and female:

```
~(?x[gender hasValue ?y] memberOf Human and ?y = Male and ?z = Female).
```

The brother of a parent is an uncle:

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

Do not trust strangers:

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

5.7. Differences between WSML-Core and WSML-Flight
The features added by WSML-Flight compared with WSML-Core are the following: Allows n-ary relations with arbitrary parameters, Constraining attribute definitions for the abstract domain, Cardinality constraints, (locally stratified) default negation in logical expressions, (in)equality in the logical language (in the body of the rule), Full-fledged rule language (based on the Datalog subset of F-Logic).
6. WSML–Rule

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

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

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

6.1. WSML–Rule Logical Expression Syntax

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

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

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

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

**Definition 6.2.** Any WSML term (see Definition 2.2) is a WSML Rule term.

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

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

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

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

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

- We define the set of admissible head formulae Head($V$) as follows:
  - Any atomic formula $\alpha$ which does not contain the inequality symbol (=) or the unification operator (=) is in Head($V$).
  - Let $\alpha, \beta \in Head(V)$, then $\alpha$ and $\beta$ is in Head($V$).
  - For $\alpha, \beta \in Head(V)$ and $\alpha, \beta$ do not contain \{ implies, impliedBy, equivalent \}, the following formulae are in Head($V$):
    - $\alpha \text{ implies } \beta$
    - $\alpha \text{ impliedBy } \beta$
    - $\alpha \text{ equivalent } \beta$
- Any admissible head formula in Head($V$) is a formula in $L(V)$.
- We define the set of admissible body formulae Body($V$) as follows:
  - Any atomic formula $\alpha$ is in Body($V$)
  - For $\alpha \in Body(V)$, nat $\alpha$ is in Body($V$).

37 of 101
- For \( \alpha, \beta \in Body(V) \), \( \alpha \text{ and } \beta \) is in \( Body(V) \).
- For \( \alpha, \beta \in Body(V) \), \( \alpha \text{ or } \beta \) is in \( Body(V) \).
- For \( \alpha, \beta \in Body(V) \), \( \alpha \text{ implies } \beta \) is in \( Body(V) \).
- For \( \alpha, \beta \in Body(V) \), \( \alpha \text{ impliedBy } \beta \) is in \( Body(V) \).
- For \( \alpha, \beta \in Body(V) \), \( \alpha \text{ equivalent } \beta \) is in \( Body(V) \).
- For variables \( ?x_1, \ldots, ?x_n \) and \( \alpha \in Body(V) \), \text{forall } \( ?x_1, \ldots, ?x_n \) is in \( Body(V) \).
- For variables \( ?x_1, \ldots, ?x_n \) and \( \alpha \in Body(V) \), \text{exists } \( ?x_1, \ldots, ?x_n \) is in \( Body(V) \).}

- Given a head–formula \( \beta \in Head(V) \) and a body–formula \( \alpha \in Body(V) \), \( \beta :- \alpha \) is a formula. Here we call \( \alpha \) the body and \( \beta \) the head of the formula. The formula is admissible if (1) \( \alpha \) is an admissible body formula, (2) \( \beta \) is an admissible head formula.
- Any formula of the form \( !- \alpha \) with \( \alpha \in Body(V) \) is an admissible formula and is called a constraint.

Any WSML–Rule admissible formula followed by a dot ‘.’ is a WSML logical expression.

As with the general WSML logical expression syntax, <–, –> and <-> can be seen as synonyms of the keywords \texttt{implies}, \texttt{impliedBy} and \texttt{equivalent}, respectively.

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

The following are examples of WSML–Rule logical expressions:

Both the father and the mother are parents:

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

Every person has a father:

\[ ?x[\text{father hasValue } f(?x)] :- ?x \text{ memberOf Person}. \]

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

\[ !- \text{distance(?location1,?location2,?distance1) and distance(?location1,?location2,?distance2) and } ?\text{distance1 } !\text{ distance2}. \]

\[ \text{distance(?A,?B,?distance) :- distance(?B,?A,?distance).} \]

6.2. Differences between WSML-Flight and WSML–Rule

WSML–Rule allows unsafe rules and the use of function symbols in the language.
7. WSML—Full

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

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

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

7.1. Differences between WSML-DL and WSML-Full

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

7.2. Differences between WSML-Rule and WSML-Full

WSML–Full adds disjunction, classical negation, multiple model semantics, and the equality operator.
8. WSML Semantics

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

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

In the following we provide first a mapping between the conceptual syntax for ontologies and the logical expression syntax for that part of the conceptual syntax which has a meaning in the logical language. We then provide a semantics for WSML–Core, WSML–Flight, WSML–Rule and WSML–DL through mapping to existing logical formalisms.

8.1. Mapping Conceptual Syntax to Logical Expression Syntax

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

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

- Introduce unnumbered anonymous identifiers for missing identifiers.
- Remove all non–functional properties from the conceptual model.
- Replace idlists with single ids for subRelationOf.
  - E.g., "P subRelationOf (Q, R)" is substituted by "P subRelationOf Q" and "P subRelationOf R".
- Expand all SQNames to full IRIs using the namespace declarations.
- Replace occurrence of an unnumbered anonymous identifier with a unique new IRI (not used in the ontology).

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

<table>
<thead>
<tr>
<th>WSML Conceptual Syntax</th>
<th>WSML Logical Expression Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ(ontology)</td>
<td>τ(ontology_element,...,ontology_element)</td>
</tr>
<tr>
<td>τ(concept id superconcept attribute, ... attribute)</td>
<td>τ(superconcept, id) τ(attribute, id) ... τ(attribute, id)</td>
</tr>
<tr>
<td>τ(subConceptOf idlist, X)</td>
<td>X subConceptOf idlist</td>
</tr>
<tr>
<td>τ(attribute_id attributefeature impliesType cardinality range_idlist, X)</td>
<td>τ(attribute_id impliesType range_idlist), τ(cardinality, X, attribute_id) τ(attributefeature, X, attribute_id)</td>
</tr>
<tr>
<td>τ(attribute_id attributefeature ofType cardinality range_idlist, X)</td>
<td>τ(attribute_id ofType range_idlist), τ(cardinality, X, attribute_id) τ(attributefeature, X, attribute_id)</td>
</tr>
<tr>
<td>τ(transitive, X, Y)</td>
<td>?x memberOf X and ?y memberOf X and ?x[ Y hasValue ?y] and ?y[ Y hasValue ?z] implies ?x[ Y hasValue ?z].</td>
</tr>
<tr>
<td>τ(symmetric, X, Y)</td>
<td>?x memberOf X and ?y memberOf X and ?x[ Y hasValue ?y] implies ?y[ Y hasValue ?x].</td>
</tr>
<tr>
<td>τ(reflexive, X, Y)</td>
<td>?x memberOf X implies ?x[ Y hasValue ?x].</td>
</tr>
</tbody>
</table>

Table 8.1: Mapping WSML conceptual syntax to logical expression syntax.
### WSML Conceptual Syntax

<table>
<thead>
<tr>
<th>WSML Conceptual Syntax</th>
<th>WSML Logical Expression Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ(inverseOf(att_id, X, Y))</td>
<td>?x memberOf X and ?x[Y hasValue ?y] implies ?y[att_id hasValue ?x].</td>
</tr>
<tr>
<td>τ(n, X, Y)</td>
<td>τ((n n), X, Y)</td>
</tr>
<tr>
<td>τ((n m), X, Y)</td>
<td>τ((n m), X, Y)</td>
</tr>
<tr>
<td>τ((n*), X, Y)</td>
<td>( \tau(p_{\text{new}}(x) \Leftarrow ?x \text{memberOf } X \land \text{Y hasValue } ?y_1 \land \ldots \land \text{Y hasValue } ?y_n). )</td>
</tr>
<tr>
<td>τ(0 m), X, Y</td>
<td>( \tau(?x \text{memberOf } X \land ?x[Y hasValue ?y_1 \land \ldots \land ?x[Y hasValue ?y_m \land \ldots \land ?y_m]). )</td>
</tr>
<tr>
<td>τ(</td>
<td>relation id</td>
</tr>
<tr>
<td>τ(</td>
<td>relation id (paramtype_1, ..., paramtype_1)</td>
</tr>
<tr>
<td>τ(</td>
<td>relation id</td>
</tr>
<tr>
<td>τ(</td>
<td>ofType idlist, X, i, n</td>
</tr>
<tr>
<td>τ(</td>
<td>impliesType idlist, X, i, n</td>
</tr>
<tr>
<td>τ(</td>
<td>subRelationOf id, X, Y</td>
</tr>
<tr>
<td>τ(</td>
<td>instance id memberof attributevalue_1, ..., attributevalue_1</td>
</tr>
<tr>
<td>τ(</td>
<td>memberOf idlist, X</td>
</tr>
<tr>
<td>τ(</td>
<td>att_id hasValue valuelist, X</td>
</tr>
<tr>
<td>τ(</td>
<td>axiom id log_expr</td>
</tr>
</tbody>
</table>

As an example, we translate the following WSML ontology:

```xml
namespace {"http://www.example.org/ontologies/example#",
dc _"http://purl.org/dc/elements/1.1#",
foaf _"http://xmlns.com/foaf/0.1",
xsd "http://www.w3.org/2001/XMLSchema#",
wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
loc _"http://www.wsmo.org/ontologies/location#",
oo _"http://example.org/ooMediator#"}
onontology Family
nfp
dc#title hasValue "WSML example ontology"
endnfp
concept Human subConceptOf (Primate, LegalAgent)
nonFunctionalProperties
dcl hasValue hasValue "concept of a human being"
endNonFunctionalProperties
hasName foaf#name
relation ageOfHuman2 (OfType Human, ofType _integer)
nfp
dc#relation hasValue FunctionalDependencyAge
endnfp
axiom FunctionalDependencyAge
definedBy
\( \tau(-\text{ageOfHuman}(x, y_1) \land \text{ageOfHuman}(x, y_2) \land \text{wsmlnumericinequal}(y_1, y_2)). \)
```

To the following logical expressions:
8.2. Preprocessing steps

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

Replacing anonymous ids with unique new identifiers

Each occurrence of an unnumbered anonymous identifier is replaced with a unique new IRI (not used in the ontology).

All occurrences of the same numbered anonymous identifier within one logical expression are replaced with the same unique new IRI (not used elsewhere in the ontology).

Replacing idlists with multiple statements

Statements involving argument lists of the form \( A \ op \ (v_1, \ldots, v_n) \), with \( op \in \{\text{hasValue}, \text{ofType}, \text{impliesType}\} \), are replaced by multiple statements in the following way: \( A \ op \ v_1, \ldots, A \ op \ v_n \).

Statements involving argument lists of the form \( A \ is-a \ \{c_1, \ldots, c_n\} \), with \( is-a \in \{\text{memberOf}, \text{subConceptOf}\} \), are replaced by a conjunction of statements in the following way: \( A \ op \ c_1 \ and \ ... \ and \ A \ op \ c_n \).

Reducing composed molecules to single molecules

Composed molecules are split into singular molecules in two steps:
- Molecules of the form \( a \ is-a \ b[c_1 \ op_1 d_1, \ldots, c_n \ op_n d_n] \) with \( is-a \in \{\text{memberOf}, \text{subConceptOf}\} \) and \( op_1 \in \{\text{hasValue}, \text{ofType}, \text{impliesType}\} \) are transformed to: \( a \ is-a b \) and \( a[c_1 \ op_1 d_1, \ldots, c_n \ op_n d_n] \).
- Then, attributes of the form \( a[c_1 \ op_1 d_1, \ldots, c_n \ op_n d_n] \), with \( op_1 \in \{\text{hasValue}, \text{ofType}, \text{impliesType}\} \), are translated to: \( a[c_1 \ op_1 d_1] \ and \ ... \ and \ a[c_n \ op_n d_n] \).

Replacing equivalence with two implications

\( \text{lexpr equivalent rexpr}. \Rightarrow \text{lexpr implies rexpr. lexpr impliedBy rexpr.} \)

Replacing right implication with left implication.

\( \text{lexpr implies rexpr}. \Rightarrow \text{rexpr impliedBy lexpr.} \)

Rewriting data term shortcuts

The shortcuts for writing strings, integers and decimals are rewritten to their full form:
- \( \text{"string"} \Rightarrow \_\text{string}("\text{string}\) \) (unless \( \text{"string"} \) already occurs in the \( \_\text{string} \) datatype wrapper)
- \( \text{integer} \Rightarrow \_\text{integer}("\text{integer}\) \)
- \( \text{decimal} \Rightarrow \_\text{decimal}("\text{decimal}\) \)

Rewrite data terms to predicates

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

Rewrite built-in functions to predicates

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

Unfolding sQNames to full IRIs

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>WSML</th>
<th>Horn logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi(head\ impliedBy$</td>
<td>$\pi(head) \leftarrow \pi(body)$</td>
</tr>
<tr>
<td>$expr$</td>
<td>$\pi(expr) \lor \pi(rexpr)$</td>
</tr>
<tr>
<td>$\pi(expr$</td>
<td>$\pi(expr) \land \pi(rexpr)$</td>
</tr>
<tr>
<td>$\pi(X1$ memberOf $id2)$</td>
<td>$id2(X1)$</td>
</tr>
<tr>
<td>$\pi(id1$ subConceptOf $id2)$</td>
<td>$id2(x) \leftarrow id1(x)$</td>
</tr>
<tr>
<td>$\pi(X1[id2 hasValue X2])$</td>
<td>$id2(X1,X2)$</td>
</tr>
<tr>
<td>$\pi(id1[id2 impliesType id3]$)</td>
<td>$id3(y) \leftarrow id1(x) \land id2(x,y)$</td>
</tr>
<tr>
<td>$\pi(id1[id2 ofType dt]))$</td>
<td>$idt(y) \leftarrow id1(x) \land id2(x,y)$</td>
</tr>
<tr>
<td>$\pi(p(X_1,\ldots,X_n))$</td>
<td>$p(X_1,\ldots,X_n)$</td>
</tr>
</tbody>
</table>

Table 8.2: WSML-Core Semantics

Note that in the translation of typing constraints using `ofType`, we introduce a (built-in) datatype predicate in the head of the rule. However, rule engines typically do not allow built-in predicates in the head of a rule. For implementation in database/rule systems, this formula can be rewritten to the following constraint: $\pi(\neg dt(y)) \leftarrow id1(x) \land id2(x,y)$, with ‘not’ being default negation. It is the same for the use of `impliesType` with a datatype as range.

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

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

- Rules of the form $A_1 \land \ldots \land A_n \leftarrow B$ are split in $n$ different rules:
  - $A_1 \leftarrow B$
  - $A_2 \leftarrow B$
  - $\ldots$
  - $A_n \leftarrow B$
- Rules of the form $A_1 \leftarrow A_2 \leftarrow B$ are transformed to:
  - $A_1 \leftarrow A_2 \land B$
- Rules of the form $A \leftarrow B_1 \lor \ldots \lor B_n$ are split into $n$ different rules:
  - $A \leftarrow B_1$
  - $\ldots$
  - $A \leftarrow B_n$

**Definition 8.2 (Satisfiability in WSML-Core)** We say a WSML-Core knowledge base $KB$ is satisfiable
iff \( \pi(KB_A) \) has a model under the semantics of first-order predicate calculus.

**Definition 8.3 (Entailment in WSML–Core)** We say a WSML–Core knowledge base \( KB_A \) entails a WSML–Core knowledge base \( KB_B \) written as: \( KB_A \models_{\text{WSML}} KB_B \) iff \( \pi(KB_A) \models \pi(KB_B) \), where \( \models \) is the classical entailment relation in first-order predicate calculus.

### 8.4. WSML-Flight Semantics

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

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

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

In our translation to F–Logic we use four kinds of molecules \( (A, B, C \text{ denote terms}) \), which we can define intuitively as:

- \( A:B \) denotes that \( A \) is a member of class \( B \).
- \( A:B \) denotes that \( A \) is a subclass of \( B \).
- \( A[B \text{ --> } C] \) denotes that object \( A \) has the value \( C \) for attribute \( B \).
- \( A[B \text{ --> } C] \) denotes that class \( A \) has an attribute \( B \) with range \( C \).

Concepts, instances and attributes are interpreted as objects in F–Logic. We need a number of auxiliary rules in order to ensure the correct interpretation of the translated F–Logic statements (with \( \text{not} \) denoting default negation under the Perfect Model Semantics [Przymusinski, 1989] and a rule with an empty head denoting an integrity constraint):

The semantics of method signatures is captured through an integrity constraint on method signatures:
\[
\leftarrow x[y \text{ --> } z] \land w:x \land w[y \text{ --> } v] \land \neg v:z
\]

The semantics of 'impliesType' is captured through an auxiliary predicate:
\[
v:z \leftarrow _{\text{impliestype}}(x,y,z) \land w:x \land w[y \text{ --> } v]
\]

Now follows the semantics of WSML–Flight in Table 8.3. In the table, \( X\# \) stands for either a variable or an identifier; '=' is the unification operator and '\(!=' is the built–in inequality symbol. The symbol \( \leftarrow_{\text{LT}} \) stands for Lloyd–Topor implication, which is eliminated from the formulae as indicated below Table 8.3.

<table>
<thead>
<tr>
<th>WSML</th>
<th>F–Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(\text{head :- body.}) )</td>
<td>( \pi(\text{head}) \leftarrow \pi(\text{body}) )</td>
</tr>
<tr>
<td>( \pi(\text{expr impliedBy expr}) )</td>
<td>( \pi(\text{expr}) \leftarrow_{\text{LT}} \pi(\text{expr}) )</td>
</tr>
<tr>
<td>( \pi(\text{expr implies expr}) )</td>
<td>( \pi(\text{expr}) \leftarrow_{\text{LT}} \pi(\text{expr}) )</td>
</tr>
<tr>
<td>( \pi(\text{expr equivalent expr}) )</td>
<td>( \pi(\text{expr}) \leftarrow_{\text{LT}} \pi(\text{expr}) \land (\pi(\text{expr}) \leftarrow_{\text{LT}} \pi(\text{expr})) )</td>
</tr>
<tr>
<td>( \pi(\text{expr or expr}) )</td>
<td>( \pi(\text{expr}) \lor \pi(\text{expr}) )</td>
</tr>
<tr>
<td>( \pi(\text{expr and expr}) )</td>
<td>( \pi(\text{expr}) \land \pi(\text{expr}) )</td>
</tr>
<tr>
<td>( \pi(\text{naf expr}) )</td>
<td>( \text{not} \pi(\text{expr}) )</td>
</tr>
<tr>
<td>( \pi(\text{X1 memberOf X2}) )</td>
<td>( \text{X1:X2} )</td>
</tr>
</tbody>
</table>

Table 8.3: Semantics of WSML–Flight
<table>
<thead>
<tr>
<th>WSML</th>
<th>F–Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi(X_1 \text{ subConceptOf } X_2)$</td>
<td>$X_1::X_2$</td>
</tr>
<tr>
<td>$\pi(X_1[X_2 \text{ hasValue } X_3])$</td>
<td>$X_1[X_2 \rightarrow X_3]$</td>
</tr>
<tr>
<td>$\pi(X_1[X_2 \text{ ofType } X_3])$</td>
<td>$X_1[X_2 \rightarrow X_3]$</td>
</tr>
<tr>
<td>$\pi(X_1[X_2 \text{ impliesType } X_3])$</td>
<td>$\text{impliestype}(X_1,X_2,X_3)$</td>
</tr>
<tr>
<td>$\pi(p(X_{i,\ldots,X_n}))$</td>
<td>$p(X_{i,\ldots,X_n})$</td>
</tr>
<tr>
<td>$\pi(X_i = X_j)$</td>
<td>$X_i = X_j$</td>
</tr>
<tr>
<td>$\pi(X_i \neq X_j)$</td>
<td>$X_i \neq X_j$</td>
</tr>
</tbody>
</table>

Rules with empty heads are integrity constraints. The first row in the table produces integrity constraints from the WSML logical expressions. Furthermore, there exists the integrity constraint which axiomatizes the semantics of `ofType`. We define $C$ as the set of all integrity constraints.

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

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

- Rules of the form $A_1 \land \ldots \land A_n \leftarrow B$ are split into $n$ different rules:
  - $A_1 \leftarrow B$
  - $\ldots$
  - $A_n \leftarrow B$

- Rules of the form $A_1 \leftarrow A_2 \land B_2$ are transformed to:
  - $A_1 \leftarrow A_2 \land B$

- Rules of the form $A \leftarrow B_1 \land (F \lor G) \land B_n$ are split into two different rules:
  - $A \leftarrow B_1 \land F \land B_n$
  - $A \leftarrow B_1 \land G \land B_n$

- Rules of the form $A \leftarrow B_1 \land \lnot (F \land G) \land B_n$ are split into two different rules:
  - $A \leftarrow B_1 \land \lnot F \land B_n$
  - $A \leftarrow B_1 \land \lnot G \land B_n$

- Rules of the form $A \leftarrow B_1 \land \lnot (F \lor G) \land B_n$ are transformed to:
  - $A \leftarrow B_1 \land \lnot F \land \lnot G \land B_n$

- Rules of the form $A \leftarrow B_1 \land \lnot F \land B_n$ are transformed to:
  - $A \leftarrow B_1 \land F \land B_n$

We base the semantics of WSML–Flight on the perfect model semantics ([Przymusinski, 1989]) of F–Logic ([Kifer et al., 1995, Appendix A]), which defines the semantics for locally stratified logic programs. Przymusinski shows that every stratified program has a unique perfect model. WSML–Flight only allows locally stratified negation.

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

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

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

### 8.5. WSML-Rule Semantics

The semantics of WSML–Rule is defined in the same way as WSML–Flight. The only difference is that the semantics of WSML–Rule is not defined through a mapping to Datalog, but through a mapping to full Logic Programming (i.e., with function symbols and allowing unsafe rules) with inequality and unstratified negation. The only difference is that the meta–variables \( X\# \) can also stand for a constructed term.

In order to define the semantics of WSML–Rule, we first define the notion of a WSML–Rule knowledge base in Definition 8.7.

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

We define the semantics of WSML–Rule through a mapping to the Horn fragment of F–Logic [Kifer et al., 1995] (extended with inequality and Well–Founded default negation [van Gelder et al., 1991] in the body of the rule) using the mapping function \( \pi \).

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

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

\[
\leftarrow x[y \rightarrow z] \land w:x \land w[y \rightarrow v] \land \neg v:z
\]

The semantics of ‘impliesType’ is captured through an auxiliary predicate:

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

Now follows the semantics of WSML–Rule in Table 8.4. In the table, \( X\# \) stands for either a variable, an identifier, or a constructed term; \( '=' \) is the unification operator and \( \not= \) is the built–in inequality symbol.
Rules with empty heads are integrity constraints (in the database sense). The first row in the table produces integrity constraints from the WSML logical expressions. Furthermore, there exists the integrity constraint which axiomatizes the semantics of `ofType`. We define `C` as the set of all integrity constraints.

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

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

- Rules of the form \( A_1 \land \ldots \land A_n \leftarrow B \) are split into \( n \) different rules:
  - \( A_1 \leftarrow B \)
  - \( \ldots \)
  - \( A_n \leftarrow B \)
- Rules of the form \( A_1 \leftarrow_{LT} A_2 \leftarrow B \) are transformed to:
  - \( A_1 \leftarrow A_2 \land B \)
- Rules of the form \( A \leftarrow B_1 \land (G \leftarrow_{LT} F) \land B_2 \) are transformed to the two rules:
  - \( A \leftarrow B_1 \land \neg F \land B_2 \)
  - \( A \leftarrow B_1 \land G \land B_2 \)
• Rules of the form \( A \leftarrow B_1 \land G \land B_2 \) are transformed to:
  \( A \leftarrow B_1 \land G \land B_2 \)

• Rules of the form \( A \leftarrow B_1 \land \forall X_1,...,X_n (G) \land B_2 \) are transformed to:
  \( A \leftarrow B_1 \land \forall X_1,...,X_n (G) \land B_2 \)

• Rules of the form \( A \leftarrow B_1 \land \exists X_1,...,X_n (G) \land B_2 \) are transformed to two rules (with \( p \) a newly
  introduced predicate symbol and \( Y_1,...,Y_k \) free variables occurring in \( \exists X_1,...,X_n (G) \)):
  \( A \leftarrow B_1 \land \exists p(Y_1,...,Y_k) \land B_2 \)
  \( p(Y_1,...,Y_k) \leftarrow G \)

• Rules of the form \( A \leftarrow B_1 \land \forall X_1,...,X_n (G) \land B_2 \) are transformed to:
  \( A \leftarrow B_1 \land \forall X_1,...,X_n (G) \land B_2 \)

• Rules of the form \( A \leftarrow B_1 \land (F \lor G) \land B_n \) are transformed to:
  \( A \leftarrow B_1 \land F \land B_2 \)
  \( A \leftarrow B_1 \land G \land B_2 \)

• Rules of the form \( A \leftarrow B_1 \land (F \land G) \land B_n \) are transformed to:
  \( A \leftarrow B_1 \land F \land B_2 \)
  \( A \leftarrow B_1 \land G \land B_2 \)

We base the semantics of WSML–Rule on the Well–Founded Semantics [van Gelder et al., 1991],
applied to F–Logic according to [Yang & Kifer, 2002].

**Definition 8.8 (Satisfiability in WSML–Rule)** Let \( KB \) be a WSML–Rule knowledge base which includes a
set of constraints \( C \). \( KB \) is satisfiable if \( \pi(KB,C) \) has well–founded partial model \( M_{KB} \)
which does not violate any of the constraints in \( C \). We say an integrity constraint is violated if the body of some ground
instantiation of the constraint is true in the model \( M_{KB} \).

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

**Definition 8.9 (Entailment in WSML–Rule)** We say a satisfiable WSML–Rule knowledge base \( KB \)
entails a WSML–Rule ground formula \( F \) iff \( M_{KB} \models \pi(F), \) where \( M_{KB} \) is the well–founded model of \( KB \).

### 8.6. WSML-DL Semantics

In order to define the semantics of WSML–DL, we first define the notion of a WSML–DL knowledge base in
Definition 8.10.

**Definition 8.10.** We define a WSML–DL knowledge base \( KB \) as a collection of formulas written in the
WSML logical expression language which are the result of application of the translation function \( \tau \) of
Table 8.1 and the preprocessing steps defined in Section 8.2 to a WSML–DL ontology.

We define the semantics of WSML–DL through a mapping to First–order logic using the mapping
function \( \pi \).

Table 8.5 presents the WSML–DL semantics through a direct mapping to function–free First–order logic
with equality. In the table, \( # \) can be any identifier, \( # \) is a datatype identifier, \( X# \) can be either a
variable or an identifier, \( Y# \) is a variable. Each occurrence of \( x \) and each occurrence of \( y \) represents a
newly introduced variable. Notice that Table 8.5 is a generalization of Table 8.2 (WSML–Core semantics); thus, the WSML–DL semantics is an extension of the WSML–Core semantics.
<table>
<thead>
<tr>
<th>WSML</th>
<th>First–order logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\pi(\text{lexpr impliedBy reexpr}))</td>
<td>(\pi(\text{lexpr}) \leftarrow \pi(\text{reexpr}))</td>
</tr>
<tr>
<td>(\pi(\text{lexpr or reexpr}))</td>
<td>(\pi(\text{lexpr}) \lor \pi(\text{reexpr}))</td>
</tr>
<tr>
<td>(\pi(\text{lexpr and reexpr}))</td>
<td>(\pi(\text{lexpr}) \land \pi(\text{reexpr}))</td>
</tr>
<tr>
<td>(\pi(\text{neg expr}))</td>
<td>(\neg \pi(\text{expr}))</td>
</tr>
<tr>
<td>(\pi(\text{forall } Y_1, ..., Y_n \text{ expr}))</td>
<td>(\forall Y_1, ..., Y_n \pi(\text{expr}))</td>
</tr>
<tr>
<td>(\pi(\text{exists } Y_1, ..., Y_n \text{ expr}))</td>
<td>(\exists Y_1, ..., Y_n \pi(\text{expr}))</td>
</tr>
<tr>
<td>(\pi(\text{X}_1 \text{ memberOf id2}))</td>
<td>(\text{id2}(\text{X}_1))</td>
</tr>
<tr>
<td>(\pi(\text{id1 subConceptOf id2}))</td>
<td>(\text{id2}(\text{x}) \leftarrow \text{id1}(\text{x}))</td>
</tr>
<tr>
<td>(\pi(\text{X}_1[\text{id2 hasValue } X_2]))</td>
<td>(\text{id2}(\text{X}_1, X_2))</td>
</tr>
<tr>
<td>(\pi(\text{id1[}\text{id2 impliesType id3}]))</td>
<td>(\text{id3}(\text{y}) \leftarrow \text{id1}(\text{x}) \land \text{id2}(\text{x}, \text{y}))</td>
</tr>
<tr>
<td>(\pi(\text{id1[}\text{id2 ofType dt}]))</td>
<td>(\text{dt}(\text{y}) \leftarrow \text{id1}(\text{x}) \land \text{id2}(\text{x}, \text{y}))</td>
</tr>
<tr>
<td>(\pi(p(\text{X}_1, ..., \text{X}_n)))</td>
<td>(p(\text{X}_1, ..., \text{X}_n))</td>
</tr>
<tr>
<td>(\pi(\text{X}_i =: X_2))</td>
<td>(\text{X}_i = X_2)</td>
</tr>
</tbody>
</table>

Table 8.5: WSML–DL Semantics

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

**Definition 8.11 (Satisfiability in WSML–DL)** We say a WSML–DL knowledge base \(KB\) is satisfiable iff \(\pi(KB_A)\) is satisfiable under the semantics of first–order predicate calculus [Enderton, 2002].

**Definition 8.12 (Entailment in WSML–DL)** We say a WSML–DL knowledge base \(KB_A\) entails a WSML–DL knowledge base \(KB_B\), written as: \(KB_A \models_{\text{WSML}} KB_B\) if \(\pi(KB_A) \models \pi(KB_B)\), where \(\models\) is the classical entailment relation.
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 parseable by standard XML parsers, but is quite unreadable for humans. This syntax is defined in Chapter 9.

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

**Mapping to OWL**  
A bi-directional mapping between (a subset of) OWL and WSML is given in Chapter 11.
9 XML Syntax for WSML

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

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

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

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

- sQNames need to be resolved to full IRIs, i.e., there will not be any namespace definitions in the XML syntax.
- in the conceptual syntax universal truth, universal falsehood and unnumbered anonymous IDs are resolved to: http://www.wsmo.org/wsml/wsml-syntax#true
  http://www.wsmo.org/wsml/wsml-syntax#false and 
  http://www.wsmo.org/wsml/wsml-syntax#anonymousID respectively. Numbered anonymous IDs are resolved as such: _#1 to http://www.wsmo.org/wsml/wsml-syntax#anonymousID1 _#2 to http://www.wsmo.org/wsml/wsml-syntax#anonymousID2 .... _#n to http://www.wsmo.org/wsml/wsml-syntax#anonymousIDn.
- Datatype identifiers are resolved to full IRIs by replacing the leading underscore '_' with the WSML namespace, according to Table C.1 in Appendix C. For example, the datatype identifier '_date' is resolved to http://www.wsmo.org/wsml/wsml-syntax#date
- Built−in functions and predicates are resolved to the corresponding full IRIs, as defined in Appendix C.
- Inside logical expressions, compound molecules are split into a conjunction of simple molecules.

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

In Table 9.1, all WSML keywords are marked bold. A,B,C stand for identifiers, D stands for a datatype identifier, \( DV_i \) stands for an integer value, \( DV_o \) stands for a decimal, and \( DV_s \) stands for a string data value. \( k,m,n \) are integer numbers. Productions in the grammar are underlined and linked to the production rules in Appendix A. Note that in the table we use the familiar sQName macro mechanism (see also Section 2.1.2) to abbreviate the IRIs of resources.
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>XML Tree</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>T ( ( wsmlVariant A ) ( definition_1, \ldots, definition_n ) )</td>
<td>&lt;wsml xmlns=&quot;http://www.wsmo.org/wsml/wsml-syntax#&quot; variant=&quot;A&quot;&gt; T(definition_1) \ldots T(definition_n) &lt;/wsml&gt;</td>
<td>Definitions are Ontology, Goal, WebService and Mediators</td>
</tr>
<tr>
<td>T ( ( namespace { N, \ldots, N_1, N_2, \ldots, N_n } ) )</td>
<td></td>
<td>Because sQnames were resolved to full IRIs during pre-processing, there is no translation to XML necessary for namespace definitions.</td>
</tr>
<tr>
<td>T ( ( ontology A ) ( header_1, \ldots, header_n ) ( ontology_element_1, \ldots, ontology_element_n ) )</td>
<td>&lt;ontology name=&quot;A&quot;&gt; T(header_1) \ldots T(header_n) T(ontology_element_1) \ldots T(ontology_element_n) &lt;/ontology&gt;</td>
<td>An ontology element represents all possible content of an ontology definition, i.e., concepts, relations, instances, ...</td>
</tr>
<tr>
<td>T ( ( concept C ) ( subConceptOf { B_1, \ldots, B_n } ) ( nfp attribute_1, \ldots, attribute_n ) )</td>
<td>&lt;concept name=&quot;C&quot;&gt; T(nfp) &lt;superConcept&gt;B_1&lt;/superConcept&gt; \ldots &lt;superConcept&gt;B_n&lt;/superConcept&gt; T(attribute_1) \ldots T(attribute_n) &lt;/concept&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( ( attributefeature_1, \ldots, attributefeature_n ofType cardinality C ) ( nfp ) )</td>
<td>&lt;attribute name=&quot;A&quot; type=&quot;constraining&quot;&gt; &lt;range&gt;C&lt;/range&gt; T(attributefeature_1) \ldots T(attributefeature_n) T(nfp) &lt;/attribute&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( ( attributefeature_1, \ldots, attributefeature_n impliesType cardinality C ) ( nfp ) )</td>
<td>&lt;attribute name=&quot;A&quot; type=&quot;inferring&quot;&gt; &lt;range&gt;C&lt;/range&gt; T(attributefeature_1) \ldots T(attributefeature_n) T(nfp) &lt;/attribute&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( ( transitive ) )</td>
<td>&lt;transitive/&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( ( symmetric ) )</td>
<td>&lt;symmetric/&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( ( reflexive ) )</td>
<td>&lt;reflexive/&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1: Mapping the WSML conceptual syntax to the WSML/XML syntax
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>XML Tree</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>T ( inverseOf(A) )</td>
<td>&lt;inverseOf type=&quot;A&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( (minCard maxCard) )</td>
<td>&lt;minCardinality&gt;minCard&lt;/minCardinality&gt; &lt;maxCardinality&gt;maxCard&lt;/maxCardinality&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( (card) )</td>
<td>&lt;minCardinality&gt;card&lt;/minCardinality&gt; &lt;maxCardinality&gt;card&lt;/maxCardinality&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( instance A memberOf C₁,...,Cₙ nfp attributevalue₁...attributevalueₙ )</td>
<td>&lt;instance name=&quot;A&quot;&gt; &lt;memberOf&gt;C₁&lt;/memberOf&gt; ... &lt;memberOf&gt;Cₙ&lt;/memberOf&gt; T(nfp) T(attributevalue₁)...T(attributevalueₙ) &lt;/instance&gt;</td>
<td>A value has always a datatype. There are four built-in datatypes: IRI, string, integer, decimal. In addition any arbitrary datatype can be defined by use of datatype wrappers. The next four transformations show how to handle these five cases.</td>
</tr>
<tr>
<td>T ( A hasValue {value₁,...,valueₙ} )</td>
<td>&lt;attributeValue name=&quot;A&quot;&gt; T(value₁)...T(valueₙ) &lt;/attributeValue&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( A )</td>
<td>&lt;value type=&quot;wsml#iri&quot;&gt; A &lt;/value&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( DV₁ )</td>
<td>&lt;value type=&quot;wsml#string&quot;&gt; DV₁ &lt;/value&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( DVᵢ )</td>
<td>&lt;value type=&quot;wsml#integer&quot;&gt; DVᵢ &lt;/value&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( DVₜ )</td>
<td>&lt;value type=&quot;wsml#decimal&quot;&gt; DVₜ &lt;/value&gt;</td>
<td>A datatype wrapper is defined by the type and a number of arguments that define the value, e.g., _date(2005,12,12) for December 12, 2005 (and thus the value of type date has three arguments). In case there is just one argument, the element &lt;argument&gt; is optional and the value can be given as above for IRI, string, integer, decimal.</td>
</tr>
<tr>
<td>T ( )</td>
<td>&lt;value type=&quot;http://www.example.org/datatype#any&quot;&gt; &lt;argument&gt;DV.arg₁&lt;/argument&gt; ... &lt;argument&gt;DV.argₙ&lt;/argument&gt; &lt;/value&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( relation A / n (paramtype₁,...,paramtypeₘ) subRelationOf C₁,...,Cₖ nfp )</td>
<td>&lt;relation name=&quot;A&quot; arity=&quot;n&quot;&gt; &lt;parameters&gt; T(paramtype₁) ... T(paramtypeₘ) &lt;/parameters&gt; &lt;superRelation&gt;C₁&lt;/superRelation&gt; &lt;/relation&gt;</td>
<td>Note that the parameters of a relation are ordered and thus the order of the parameters elements in the XML representation important.</td>
</tr>
<tr>
<td>WSML syntax</td>
<td>XML Tree</td>
<td>Remarks</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>...&lt;superRelation&gt;C_k&lt;/superRelation&gt; T(nfp) &lt;/relation&gt;...</td>
<td>&lt;parameter type=&quot;constraining&quot;&gt; &lt;range&gt;C_i&lt;/range&gt; ... &lt;range&gt;C_n&lt;/range&gt; &lt;/parameter&gt;</td>
<td></td>
</tr>
<tr>
<td>T(oOfType {C_1,...,C_n})</td>
<td>&lt;parameter type=&quot;inferring&quot;&gt; &lt;range&gt;C_i&lt;/range&gt; ... &lt;range&gt;C_n&lt;/range&gt; &lt;/parameter&gt;</td>
<td></td>
</tr>
<tr>
<td>T(impliesType {C_1,...,C_n})</td>
<td>&lt;relationInstance name=&quot;A&quot;&gt; &lt;memberOf&gt;B&lt;/memberOf&gt; T(value_1) ... T(value_n) T(nfp) &lt;/relationInstance&gt;</td>
<td></td>
</tr>
<tr>
<td>T(relationInstance A B (value_1,...,value_n) nfp)</td>
<td>&lt;parameterValue&gt; T(value_1) ... T(value_n) &lt;/parameterValue&gt; Note that the parameters of a relationInstance are ordered and thus the order of the value elements is important.</td>
<td></td>
</tr>
<tr>
<td>T( axiom A axiomdefinition )</td>
<td>&lt;axiom name=&quot;A&quot;&gt; T( axiomdefinition ) &lt;/axiom&gt;</td>
<td></td>
</tr>
<tr>
<td>T(nfp definedBy log_expr)</td>
<td>T(nfp) &lt;definedBy&gt; T(log_expr) &lt;/definedBy&gt; The mapping of logical expressions is defined in Table 9.2.</td>
<td></td>
</tr>
<tr>
<td>T(goal A header_1 ... header_n capability interface_1 ... interface_n)</td>
<td>&lt;goal name=&quot;A&quot;&gt; T(header_1) ... T(header_n) T(capability) T(interface_1) ... T(interface_n) &lt;/goal&gt;</td>
<td></td>
</tr>
<tr>
<td>T(ooMediator A nfp importsontology source target use_service)</td>
<td>&lt;ooMediator name=&quot;A&quot;&gt; T(nfp) T( importsontology ) T(source) T(target) T(use_service) &lt;/ooMediator&gt;</td>
<td></td>
</tr>
<tr>
<td>T(ggMediator A header_1 ... )</td>
<td>&lt;ggMediator name=&quot;A&quot;&gt; T(header_1) ... T(header_n)</td>
<td></td>
</tr>
<tr>
<td>WSML syntax</td>
<td>XML Tree</td>
<td>Remarks</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td><code>header</code></td>
<td><code>T(source)</code></td>
<td></td>
</tr>
<tr>
<td><code>source</code></td>
<td><code>T(target)</code></td>
<td></td>
</tr>
<tr>
<td><code>target</code></td>
<td><code>T(use_service)</code></td>
<td></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>&lt;/ggMediator&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T (wgMediator A)</code></td>
<td><code>&lt;wgMediator name=&quot;A&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>header</code></td>
<td><code>T(header_1)</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>T(header_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>source</code></td>
<td><code>source</code></td>
<td></td>
</tr>
<tr>
<td><code>target</code></td>
<td><code>T(source)</code></td>
<td></td>
</tr>
<tr>
<td><code>use_service</code></td>
<td><code>T(target)</code></td>
<td></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>T(use_service)</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/wgMediator&gt;</code></td>
<td><code>&lt;/wgMediator&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T (wwMediator A)</code></td>
<td><code>&lt;wwMediator name=&quot;A&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>header</code></td>
<td><code>T(header_1)</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>T(header_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>source</code></td>
<td><code>source</code></td>
<td></td>
</tr>
<tr>
<td><code>target</code></td>
<td><code>T(source)</code></td>
<td></td>
</tr>
<tr>
<td><code>use_service</code></td>
<td><code>T(target)</code></td>
<td></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>T(use_service)</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/wwMediator&gt;</code></td>
<td><code>&lt;/wwMediator&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T (source { A_1, ..., A_n })</code></td>
<td><code>&lt;source&gt;A_1&lt;/source&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>&lt;source&gt;A_n&lt;/source&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>T (target A)</code></td>
<td><code>&lt;target&gt;A&lt;/target&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>T (usesService A)</code></td>
<td><code>&lt;usesService&gt;A&lt;/usesService&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>T (webService A)</code></td>
<td><code>&lt;webService name=&quot;A&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>header</code></td>
<td><code>T(header_1)</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>T(header_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>capability</code></td>
<td><code>T(capability)</code></td>
<td></td>
</tr>
<tr>
<td><code>interface</code></td>
<td><code>T(interface_1)</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>T(interface_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>&lt;/webService&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T (capability C)</code></td>
<td><code>&lt;capability name=&quot;C&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>header</code></td>
<td><code>T(header_1)</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>T(header_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>sharedvardef</code></td>
<td><code>T(sharedvardef)</code></td>
<td></td>
</tr>
<tr>
<td><code>pre_post_ass_or_eff</code></td>
<td><code>T(pre_post_ass_or_eff_1)</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>T(pre_post_ass_or_eff_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>&lt;/capability&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T (sharedVariables {?v_1, ..., ?v_n})</code></td>
<td><code>&lt;sharedVariables&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>&lt;variable name=&quot;?v_1&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;variable name=&quot;?v_n&quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/sharedVariables&gt;</code></td>
<td><code>&lt;/sharedVariables&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

*pre_post_ass_or_eff* unites the axiom definitions for precondition, assumption, postcondition and effect.
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>XML Tree</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>T ( precondition $B$ axiomdefinition )</td>
<td>&lt;precondition name=&quot;B&quot;&gt; T(axiomdefinition) &lt;/precondition&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( assumption $B$ axiomdefinition )</td>
<td>&lt;assumption name=&quot;B&quot;&gt; T(axiomdefinition) &lt;/assumption&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( postcondition $B$ axiomdefinition )</td>
<td>&lt;postcondition name=&quot;B&quot;&gt; T(axiomdefinition) &lt;/postcondition&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( effect $B$ axiomdefinition )</td>
<td>&lt;effect name=&quot;B&quot;&gt; T(axiomdefinition) &lt;/effect&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( interface $A$ header$_1$ ... header$_n$ choreography orchestration )</td>
<td>&lt;interface name=&quot;A&quot;&gt; T(header$_1$) ... T(header$_n$) T(choreography) T(orchestration) &lt;/interface&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( choreography $C$ )</td>
<td>&lt;choreography&gt;$C$&lt;/choreography&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( orchestration $A$ )</td>
<td>&lt;orchestration&gt;$A$&lt;/orchestration&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( nonFunctionalProperties attributevalue$_1$ ... attributevalue$_n$ endNonFunctionalProperties )</td>
<td>&lt;nonFunctionalProperties&gt; T(attributevalue$_1$) ... T(attributevalue$_n$) &lt;/nonFunctionalProperties&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( nfp attributevalue$_1$ ... attributevalue$_n$ endnfp )</td>
<td>&lt;nonFunctionalProperties&gt; T(attributevalue$_1$) ... T(attributevalue$_n$) &lt;/nonFunctionalProperties&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( importsOntology { $A_1$,...,$A_n$ } )</td>
<td>&lt;importsOntology&gt;$A_1$&lt;/importsOntology&gt; ... &lt;importsOntology&gt;$A_n$&lt;/importsOntology&gt;</td>
<td></td>
</tr>
<tr>
<td>T ( usesMediator { $B_1$,...,$B_n$ } )</td>
<td>&lt;usesMediator&gt;$B_1$&lt;/usesMediator&gt; ... &lt;usesMediator&gt;$B_n$&lt;/usesMediator&gt;</td>
<td></td>
</tr>
</tbody>
</table>

The logical expression syntax is explained in Table 9.2. In the table, $A$ stands for identifiers and $T_1$,...,$T_n$ stand for terms. $V$ stands for a variable.
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>XML Tree</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| $T' (A(term_1, ..., term_n))$ | `<term name="A"> T' (term_1, arg) ...
T' (term_n, arg) </term>` | |
| $T' (A(term_1, ..., term_n), name)$ | `<name name="A"> T' (term_1, arg) ...
T' (term_n, arg) </name>` | |
| $T (V)$ | `<term name="V"/>` | |
| $T (V, name)$ | `<name name="V"> </name>` | |
| $T (A(term_1, ..., term_n))$ | `<atom name="A"> T' (term_1, arg) ...
T' (term_n, arg) </atom>` | |
| $T (T_1[attr_relation_1, ..., attr_relation_n] subConceptOf T_2, ..., T_n)$ | `<molecule>`
`T'(T_1)`
`<isa type="subConceptOf"> T'(T_2)`
`... T'(T_n) </isa>`
`T(attr relation_1)`
`... T(attr relation_n) </molecule>` | |
| $T (T_1[attr_relation_1, ..., attr_relation_n] memberOf T_2, ..., T_n)$ | `<molecule>`
`T'(T_1)`
`<isa type="memberOf"> T'(T_2)`
`... T'(T_n) </isa>`
`T(attr relation_1)`
`... T(attr relation_n) </molecule>` | |
| $T (T_0 ofType \{T_1, ..., T_n\})$ | `<molecule>`
`T(T_0)`
`T(attr relation_1)`
`... T(attr relation_n) </molecule>` | |
| $T (T_0 ofType \{T_1, ..., T_n\})$ | `<attributeDefinition type="constraining"> T'(T_0, name)
T'(T_1, type)
... T'(T_n, type) </attributeDefinition>` | |

Table 9.2: [TODO: make distinction between expressions and terms in translation function] Mapping the WSML syntax into XML Tree.
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>XML Tree</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T ( \impliesType { T_0, \ldots, T_n } ) )</td>
<td>(&lt;\text{attributeDefinition type=&quot;inferring&quot;&gt;} \ T(T_0, \text{name}) \ T(T_1, \text{type}) \ T(T_n, \text{type}) \ &lt;\text{/attributeDefinition}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \hasValue { T_0, \ldots, T_n } ) )</td>
<td>(&lt;\text{attributeValue}&gt; \ T(T_0, \text{name}) \ T(T_1, \text{value}) \ T(T_n, \text{value}) \ &lt;\text{/attributeValue}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \text{and} \ expr_1, \ expr_2 ) )</td>
<td>(&lt;\text{and}&gt; \ T(expr_1) \ T(expr_2) \ &lt;\text{/and}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \text{or} \ expr_1, \ expr_2 ) )</td>
<td>(&lt;\text{or}&gt; \ T(expr_1) \ T(expr_2) \ &lt;\text{/or}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \neg \ expr ) )</td>
<td>(&lt;\text{neg}&gt; \ T(expr) \ &lt;\text{/neg}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \text{naf} \ expr ) )</td>
<td>(&lt;\text{naf}&gt; \ T(expr) \ &lt;\text{/naf}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \implies \ expr_1, \ expr_2 ) )</td>
<td>(&lt;\text{implies}&gt; \ T(expr_1) \ T(expr_2) \ &lt;\text{/implies}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \impliedBy \ expr_1, \ expr_2 ) )</td>
<td>(&lt;\text{impliedBy}&gt; \ T(expr_1) \ T(expr_2) \ &lt;\text{/impliedBy}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \equiv \ expr_1, \ expr_2 ) )</td>
<td>(&lt;\text{equivalent}&gt; \ T(expr_1) \ T(expr_2) \ &lt;\text{/equivalent}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \forall \ ?v_1, \ldots, \ ?v_n \ ( \text{forall} \ forall \ ?v_1, \ldots, \ ?v_n \ ( \ expr ) ) )</td>
<td>(&lt;\text{forall}&gt; \ &lt;\text{variable name=&quot;?v_1&quot;/&gt;} \ T(expr) \ &lt;\text{/forall}&gt; )</td>
<td></td>
</tr>
<tr>
<td>( T ( \exists \ ?v_1, \ldots, \ ?v_n \ ( \text{exists} \ exists \ ?v_1, \ldots, \ ?v_n \ ( \ expr ) ) )</td>
<td>(&lt;\text{exists}&gt; \ &lt;\text{variable name=&quot;?v_1&quot;/&gt;} \ T(expr) \ &lt;\text{/exists}&gt; )</td>
<td></td>
</tr>
<tr>
<td>WSML syntax</td>
<td>XML Tree</td>
<td>Remarks</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| $T \ ( \ \!− \ expr \ )$ | `<constraint>`  
$T(expr)$  
`</constraint>` | |
| $T \ ( expr_1 :− expr_2 \ )$ | `<impliedByLP>`  
$T(expr_1)$  
$T(expr_2)$  
`</impliedByLP>` | |

Table 9.3 provides a simple translation example.
<table>
<thead>
<tr>
<th>WSML Syntax</th>
<th>XML Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsmlVariant _http://www.wsmo.org/wsml/wsml-syntax/wsml-flight</code></td>
<td></td>
</tr>
<tr>
<td><code>namespace _http://www.example.org/ex1</code>, wsmi _<a href="http://www.wsmo.org/wsml/wsml-syntax#">http://www.wsmo.org/wsml/wsml-syntax#</a>, ex _<a href="http://www.example.org/ex2#%60">http://www.example.org/ex2#`</a></td>
<td></td>
</tr>
<tr>
<td><code>ontology _http://www.example.org/ex1</code></td>
<td></td>
</tr>
<tr>
<td><code>nonFunctionalProperties</code></td>
<td></td>
</tr>
<tr>
<td><code>dc#title</code> hasValue &quot;WSML to RDF&quot;</td>
<td></td>
</tr>
<tr>
<td><code>dc#date</code> hasValue <code>date(2005,12,12)</code></td>
<td></td>
</tr>
<tr>
<td><code>endNonFunctionalProperties</code></td>
<td></td>
</tr>
<tr>
<td><code>importsOntology _http://www.example.net/ex2</code></td>
<td></td>
</tr>
<tr>
<td><code>concept Woman subConceptOf (ex#Human, ex#LivingBeing)</code></td>
<td></td>
</tr>
<tr>
<td><code>name ofType string</code></td>
<td></td>
</tr>
<tr>
<td><code>age ofType (1) integer</code></td>
<td></td>
</tr>
<tr>
<td><code>axiom GenderConstraint</code></td>
<td></td>
</tr>
<tr>
<td><code>definedBy</code> !- ?x memberOf ex#Man and ?x memberOf Woman.`</td>
<td></td>
</tr>
<tr>
<td><code>instance Mary memberOf Woman</code></td>
<td></td>
</tr>
<tr>
<td><code>age hasValue 23</code></td>
<td></td>
</tr>
<tr>
<td><code>relation childOf/2 (ofType ex#Human, impliesType ex#Parent)</code></td>
<td></td>
</tr>
<tr>
<td><code>webService ws</code></td>
<td></td>
</tr>
<tr>
<td><code>capability itineraryInfo</code></td>
<td></td>
</tr>
<tr>
<td><code>interface (_http://example.org/i1&quot;, _http://example.org/i2&quot;)</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.3: A Mapping Example
10. RDF Syntax for WSML

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

<table>
<thead>
<tr>
<th>WSML keyword</th>
<th>Used as predicate to indicate the WSML variant applied.</th>
<th>Used to define a WSML goal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsml#variant</td>
<td></td>
<td>wsml#goal</td>
</tr>
<tr>
<td>wsml#ontology</td>
<td>Used to define a WSML ontology.</td>
<td>wsml#ooMediator</td>
</tr>
<tr>
<td>wsml#hasConcept</td>
<td>Used to type a WSML concept and to bind it to an ontology.</td>
<td>wsml#ggMediator</td>
</tr>
<tr>
<td>wsml#attribute</td>
<td>Used to define a WSML attribute</td>
<td>wsml#wgMediator</td>
</tr>
<tr>
<td>wsml#oType</td>
<td>Used as predicate to define constraining attributes and parameters.</td>
<td>wsml#wwMediator</td>
</tr>
<tr>
<td>wsml#hasAttribute</td>
<td>Used as predicate to bind an attribute to a concept.</td>
<td>wsml#webService</td>
</tr>
<tr>
<td>wsml#transitiveAttribute</td>
<td>Used to indicate the transitivity of an attribute.</td>
<td>wsml#useInterface</td>
</tr>
<tr>
<td>wsml#symmetricAttribute</td>
<td>Used to indicate the symmetry of an attribute.</td>
<td>wsml#useCapability</td>
</tr>
<tr>
<td>wsml#reflexiveAttribute</td>
<td>Used to indicate the reflexivity of an attribute.</td>
<td>wsml#sharedVariables</td>
</tr>
<tr>
<td>wsml#inverseOf</td>
<td>Used to indicate the inverse relationship of two attributes.</td>
<td>wsml#precondition</td>
</tr>
<tr>
<td>wsml#minCardinality</td>
<td>Used as predicate to define the minimal cardinality of an attribute.</td>
<td>wsml#assumption</td>
</tr>
<tr>
<td>wsml#maxCardinality</td>
<td>Used as predicate to define the maximal cardinality of an attribute.</td>
<td>wsml#postcondition</td>
</tr>
<tr>
<td>wsml#hasInstance</td>
<td>Used to type an instance and to bind it to a concept.</td>
<td>wsml#effect</td>
</tr>
<tr>
<td>wsml#hasRelation</td>
<td>Used to type a relation and to bind it to an ontology.</td>
<td>wsml#choreography</td>
</tr>
<tr>
<td>wsml#arity</td>
<td>Used to define the arity of a WSML relation.</td>
<td>wsml#orchestration</td>
</tr>
<tr>
<td>wsml#param</td>
<td>Used to type parameters of WSML relations.</td>
<td>wsml#nfp</td>
</tr>
<tr>
<td>wsml#subRelationOf</td>
<td>Used as predicate to define sub-relations.</td>
<td>wsml#importsOntology</td>
</tr>
<tr>
<td>wsml#hasRelationInstance</td>
<td>Used to type a relation instance and to bind it to an ontology.</td>
<td>wsml#usesMediator</td>
</tr>
<tr>
<td>wsml#hasAxiom</td>
<td>Used to type an axiom and to bind it to an ontology.</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: WSML vocabulary extending RDFS for WSML triple syntax
The remainder of this chapter presents a mapping between the human-readable syntax of WSML and the RDF/WSML-triple syntax for all WSML variants, where the knowledge is encoded in `<subject><predicate><object>`-triples. The big advantage of having such a syntax and reusing the RDFS-vocabulary as far as possible, is the fact that there are many existing RDF(S)-based tools available. These tools are optimized to handle triples and are thus able to understand parts of our specification and in a more general sense we can guarantee inter-operability with those. In RDF all triples are global, while in the conceptual syntax of WSML it is possible to append specific entities to higher-level entities, e.g., concepts to ontologies or attribute values to instances.

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

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

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

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

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

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

A simple example of a full translation from the human-readable syntax to the RDF triple notation is given in Table 10.3.
Table 10.2: Mapping to the WSML/RDF syntax

<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>RDF Triples</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>T (</td>
<td>T(definition₁, A)</td>
<td>definitions are Ontology, Goal, WebService and Mediators</td>
</tr>
<tr>
<td>wsmlVariant A</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>definition₁</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>definitionₙ</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>namespace { N,</td>
<td>A rdf#type wsml#ontology</td>
<td>Because sQnames were resolved to full IRIs during pre-processing, there is no translation to RDF necessary for namespace definitions</td>
</tr>
<tr>
<td>P₁, N₁</td>
<td>A wsml#variant Z</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(header₁, A)</td>
<td></td>
</tr>
<tr>
<td>Pₙ, Nₙ }</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T(headerₙ, A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T(ontology_element₁, A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T(ontology_elementₙ, A)</td>
<td></td>
</tr>
<tr>
<td>T (</td>
<td>A rdfs#subClassOf B₁</td>
<td></td>
</tr>
<tr>
<td>ontology A</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>header₁</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(headerₙ, A)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(ontology_element₁, A)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(ontology_elementₙ, A)</td>
<td></td>
</tr>
<tr>
<td>T (</td>
<td>A rdfs#subClassOf B₁</td>
<td></td>
</tr>
<tr>
<td>concept A</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>subConceptOf {B₁,...,Bₙ}</td>
<td>A rdf#type wsml#ontology</td>
<td></td>
</tr>
<tr>
<td>nfp attribute₁</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(attribute₁, A)</td>
<td></td>
</tr>
<tr>
<td>nfp attributeₙ</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>. Z)</td>
<td>T(attributeₙ, A)</td>
<td></td>
</tr>
<tr>
<td>T (</td>
<td>Z wsml#hasConcept A</td>
<td></td>
</tr>
<tr>
<td>A attributefeature₁</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(attributefeature₁, _X)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(attributefeatureₙ, _X)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(nfp, _X)</td>
<td>The blank identifier_:X denotes a helper node to bind the attribute A to a defined owner: attributes are locally defined!</td>
</tr>
<tr>
<td>A attributefeatureₙ, nfp</td>
<td>Z wsml#hasAttribute _:X</td>
<td></td>
</tr>
<tr>
<td>. Z)</td>
<td>T(cardinality, _X)</td>
<td></td>
</tr>
<tr>
<td>_X wsml#attribute A</td>
<td>T(nfp, _X)</td>
<td></td>
</tr>
<tr>
<td>A attributefeature₁, _X</td>
<td>T(nfp, _X)</td>
<td></td>
</tr>
<tr>
<td>Z wsml#hasAttribute _:X</td>
<td>T(nfp, _X)</td>
<td></td>
</tr>
<tr>
<td>A attributefeatureₙ, _X</td>
<td>T(nfp, _X)</td>
<td></td>
</tr>
<tr>
<td>_X wsml#attribute A</td>
<td>Z wsml#inverseOf A</td>
<td></td>
</tr>
<tr>
<td>T(attributefeature₁, _X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(attributefeatureₙ, _X)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>T(nfp, _X)</td>
<td></td>
</tr>
<tr>
<td>_X wsml#attribute A</td>
<td>T( Transitive Attribute )</td>
<td></td>
</tr>
<tr>
<td>T(transitive , Z)</td>
<td></td>
<td>T(symmetric , Z)</td>
</tr>
<tr>
<td>T(reflexive , Z)</td>
<td>Z rdf#type wsml#symmetricAttribute</td>
<td></td>
</tr>
<tr>
<td>T( inverseOf(A) , Z)</td>
<td>Z rdf#type wsml#reflexiveAttribute</td>
<td></td>
</tr>
<tr>
<td>T(inverseOf(A) , Z)</td>
<td>Z wsml#inverseOf A</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.2: Mapping to the WSML/RDF syntax
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>RDF Triples</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| $T((m,n),Z)$ | $Z\text{wsml#minCardinality }m$  
$Z\text{wsml#maxCardinality }n$ |         |
| $T((\text{card}),Z)$ | $Z\text{wsml#minCardinality card}$  
$Z\text{wsml#maxCardinality card}$ |         |
| $T(\text{instance }A)$  
$\text{memberOf }C_1,\ldots,C_n$  
$nfp$  
$\text{attributevalue}_1$  
$\ldots$  
$\text{attributevalue}_n$  
$Z$ | $Z\text{wsml#hasInstance }A$  
$A\text{rdf#type }C_1$  
$\ldots$  
$A\text{rdf#type }C_n$$T(nfp,A)$  
$T(\text{attributevalue}_1,A)$  
$\ldots$  
$T(\text{attributevalue}_n,A)$ | It is not required to associate an instance with an identifier. In that case the identifier $J$ is replaced by a blank node identifier, e.g. `$:_X$`. The same counts for all entities that do not require an ID: instance, relationInstance, but also goal, mediators, webService and capability and interface definitions. |
| $T(\text{dc#title hasValue value},Z)$ | $Z\text{rdfs#label }T(\text{value})$ |         |
| $T(\text{dc#description hasValue value},Z)$ | $Z\text{rdfs#comment }T(\text{value})$ |         |
| $T(\text{dc#relation hasValue value},Z)$ | $Z\text{rdfs#seeAlso }T(\text{value})$ |         |
| $T(A\text{hasValue value},Z)$ | $ZA T(\text{value})$ | Strings are already enclosed with double quotes in WSML; these do not have to be added for the RDF literals. |
| $T(DV_s)$ | $DV_s^{^\text{xsd#string}}$ |         |
| $T(DV_i)$ | $"DV_i"^{^\text{xsd#integer}}$ |         |
| $T(DV_d)$ | $"DV_d"^{^\text{xsd#decimal}}$ |         |
| $T(D(a_1,\ldots,a_n))$ | $T\text{serialize}(D(a_1,\ldots,a_n))^{^\text{T datatypes}(D)}$ | $T\text{serialize}$ serializes the WSML representation of a data value to a string representation which can be readily used in RDF literals. This function is not yet defined. $T\text{datatypes}$ maps WSML datatypes to XML Schema datatypes, according to Table C.1 in Appendix C. |
| $T(A)$ | $A$ | IRLs are directly used in RDF. |
| $T(\text{relation }A/n (B_1,\ldots,B_m))$  
$\text{subRelationOf }C_1,\ldots,C_k$  
$nfp$  
$Z$ | $Z\text{wsml#hasRelation }A$  
$A\text{wsml#arity }^"n"{^\text{xsd#integer}}$  
$A\text{wsml#param }:_X$  
$:_X\text{rdf#type rdf#List}$  
$:_X\text{rdf#first }:_X_1$  
$T(B_1,\ldots:_X_1)$  
$:_X\text{rdf#rest }:_1$  
$:_1\text{rdf#type rdf#List}$  
$:_1\text{rdf#first }:_X_2$ | The parameters of a relation are unnamed and thus ordered. The ordering in RDF is provided by use of the rdf#List. |
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>RDF Triples</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_m rdf#type rdf#List</code></td>
<td><code>T(_,_,_:X)</code></td>
<td></td>
</tr>
<tr>
<td><code>_m rdf#first _:X</code></td>
<td><code>R wsml#subRelationOf C_i</code></td>
<td></td>
</tr>
<tr>
<td><code>T(B_m,_,_:X)</code></td>
<td><code>R wsml#subRelationOf C_k</code></td>
<td></td>
</tr>
<tr>
<td><code>T(nfp, A)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T(ofType C, Z)</code></td>
<td><code>Z wsml#ofType C</code></td>
<td></td>
</tr>
<tr>
<td><code>T(impliesType C, Z)</code></td>
<td><code>Z rdfs#range C</code></td>
<td></td>
</tr>
<tr>
<td><code>T(relationInstance A B (valuelist) nfp, Z)</code></td>
<td><code>Z wsml#hasRelationInstance A A rdf#type B T(valuelist, A)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>T(nfp, A)</code></td>
<td></td>
</tr>
<tr>
<td><code>T(value_1,...,value_n, Z)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T( axiom A axiomdefinition, Z )</code></td>
<td><code>Z wsml#hasAxiom A T(axiomdefinition, A)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>T(nfp, A)</code></td>
<td></td>
</tr>
<tr>
<td><code>T( nfp definedBy log_expr )</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T(goal A header_1 ... header_n capability interface_1 ... interface_n, Z)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T(ooMediator A nfp importsontology source target use_service, Z)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>T(ggMediator A nfp)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The arguments of a relationinstance are unnamed and thus ordered. The ordering in RDF is provided by use of the rdf#List.

*The logical expressions are translated to literals of type rdf#XMLLiteral using the mapping function defined in Table 9.2.*
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>RDF Triples</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| ```wsml importsontology source target use_service . Z) T (                      | ```A wsmi#variant Z
A rdf#type wsmi#wgMediator
T(nfp, A)
T(importsontology, A)
T(source, A)
T(target, A)
T(use_service, A)
```                                                                                                                                 |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
| `T ( wgMediator A nfp importsontology source target use_service . Z) T (       | ```A wsmi#variant Z
A rdf#type wsmi#wMediator
T(nfp, A)
T(importsontology, A)
T(source, A)
T(target, A)
T(use_service, A)
```                                                                                                                                 |**pre_post_ass_or_eff reunites the axiom definitions for precondition, assumption, postcondition and effect.** |
| `T ( source { A_1, ..., A_n } . Z) T (                                       | ```Z wsmi#source A_1
... Z wsmi#source A_n```                                                                                                                                                                               |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
| `T ( target A . Z) T (                                                     | ```Z wsmi#target A```                                                                                                                                                                                      |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
| `T ( usesService A . Z) T (                                                 | ```Z wsmi#usesService A```                                                                                                                                                                                 |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
| `T ( webService A header_1 ... header_n capability interface_1 ... interface_n . Z) T (`       | ```A wsmi#variant Z
A rdf#type wsmi#webService
T(header_1, A)
T(header_n, A)
T(capability, A)
T(interface_1, A)
T(interface_n, A)```                                                                                                                                 |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
| `T ( capability C header_1 ... header_n sharedvardef pre_post_ass_or_eff_1 ... pre_post_ass_or_eff_n . Z) T (`       | ```Z wsmi#useCapability C
T(header_1, C)
T(header_n, C)
T(sharedvardef, C)
T(pre_post_ass_or_eff_1, C)
T(pre_post_ass_or_eff_n, C)```                                                                                                                                 |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
| `T ( sharedVariables { ?v_1, ..., ?v_n } . Z) T (                          | ```Z wsmi#sharedVariables ?v_1 ... Z wsmi#sharedVariables ?v_n```                                                                                                                                               |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
| `T ( precondition B axiomdefinition . Z) T (                                 | ```Z wsmi#precondition B
T(axiomdefinition, B)```                                                                                                                                                                                |**Please note that instead of simply using a triple per shared variable it is possible to apply the rdf#Bag container to better describe the group character of shared variables.** |
<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>RDF Triples</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(\text{assumption } B, \text{axiomdefinition }, Z) )</td>
<td>( T(\text{axiomdefinition, } B) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{postcondition } B, \text{axiomdefinition }, Z) )</td>
<td>( T(\text{axiomdefinition, } B) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{effect } B, \text{axiomdefinition }, Z) )</td>
<td>( T(\text{axiomdefinition, } B) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{interface } A \text{ header}_1 \ldots \text{header}_n \text{choreography \ orchestration}, Z) )</td>
<td>( T(\text{header}_1, A) ) \ldots ( T(\text{header}_n, A) ) ( T(\text{choreography, } A) ) ( T(\text{orchestration, } A) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{choreography } C, Z) )</td>
<td>( T(\text{choreography } C) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{orchestration } A, Z) )</td>
<td>( T(\text{orchestration } A) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{nonFunctionalProperties attributevalue}_1 \ldots \text{attributevalue}_n \text{endNonFunctionalProperties}, Z) )</td>
<td>( T(\text{attributevalue}_1, _P_1) \ldots ( T(\text{attributevalue}_n, _P_n) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{nfp attributevalue}_1 \ldots \text{attributevalue}_n \text{endnfp}, Z) )</td>
<td>( T(\text{attributevalue}_1, _P_1) \ldots ( T(\text{attributevalue}_n, _P_n) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{importsOntology} { A_1, \ldots, A_n }, Z) )</td>
<td>( T(\text{importsOntology } A_1) \ldots ( T(\text{importsOntology } A_n) )</td>
<td></td>
</tr>
<tr>
<td>( T(\text{usesMediator} { B_1, \ldots, B_n }, Z) )</td>
<td>( T(\text{usesMediator } B_1) \ldots ( T(\text{usesMediator } B_n) )</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.3 provides a simple translation example.
### WSML syntax

<table>
<thead>
<tr>
<th>WSML syntax</th>
<th>RDF Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsmlVariant</code> <em>&quot;<a href="http://www.wsmo.org/wsml/wsml-syntax/wsml-flight">http://www.wsmo.org/wsml/wsml-syntax/wsml-flight</a>&quot;</em></td>
<td>The first RDF graph for the ontology:</td>
</tr>
<tr>
<td><code>ontology</code> <em>&quot;<a href="http://www.example.org/ex1">http://www.example.org/ex1</a>&quot;</em></td>
<td><a href="http://www.example.org/ex1">http://www.example.org/ex1</a> rdf#type wsml#ontology</td>
</tr>
<tr>
<td><code>nonFunctionalProperties</code> <em>dc#title hasValue &quot;WSML to RDF&quot;</em></td>
<td><a href="http://www.example.org/ex1">http://www.example.org/ex1</a> rdf#type wsml#ontology</td>
</tr>
<tr>
<td></td>
<td>dc#date hasValue <em>date(2005,12,12)</em></td>
</tr>
<tr>
<td><code>importsOntology</code> <em>&quot;<a href="http://www.example.net/ex2">http://www.example.net/ex2</a>&quot;</em></td>
<td><a href="http://www.example.org/ex1">http://www.example.org/ex1</a> rdf#type wsml#ontology</td>
</tr>
<tr>
<td><code>concept Woman subConceptOf {ex#Human, ex#LivingBeing}</code></td>
<td><a href="http://www.example.org/ex1">http://www.example.org/ex1</a> rdfs#subClassOf <a href="http://www.example.org/ex2#Human">http://www.example.org/ex2#Human</a></td>
</tr>
<tr>
<td><code>name ofType _string ancestorOf transitiveImpliesType ex#Human</code></td>
<td><a href="http://www.example.org/ex1">http://www.example.org/ex1</a> rdfs#subClassOf <a href="http://www.example.org/ex2#LivingBeing">http://www.example.org/ex2#LivingBeing</a></td>
</tr>
<tr>
<td><code>axiom GenderConstraint definedBy !- ?x memberOf ex#Man and _?x memberOf Woman._</code></td>
<td><a href="http://www.example.org/ex1">http://www.example.org/ex1</a> rdfs#isDefinedBy &quot;&lt;constraint&gt;...&lt;/constraint&gt;&quot;^^rdf#XMLLiteral</td>
</tr>
<tr>
<td><code>instance Mary memberOf Woman</code></td>
<td><a href="http://www.example.org/ex1#WomanDef">http://www.example.org/ex1#WomanDef</a> rdf#isDefinedBy &quot;&lt;constraint&gt;...&lt;/constraint&gt;&quot;^^rdf#XMLLiteral</td>
</tr>
<tr>
<td><code>age hasValue 23</code></td>
<td><a href="http://www.example.org/ex1#WomanDef">http://www.example.org/ex1#WomanDef</a> rdf#isDefinedBy &quot;&lt;constraint&gt;...&lt;/constraint&gt;&quot;^^rdf#XMLLiteral</td>
</tr>
<tr>
<td><code>relation childOfOf2 (oftype ex#Human, impliesType ex#Parent)</code></td>
<td><a href="http://www.example.org/ex1#WomanDef">http://www.example.org/ex1#WomanDef</a> rdf#isDefinedBy &quot;&lt;constraint&gt;...&lt;/constraint&gt;&quot;^^rdf#XMLLiteral</td>
</tr>
<tr>
<td><code>webService ws capability itineraryInfo interface (_&quot;http://example.org/i1&quot;, _&quot;http://example.org/i2&quot;)</code></td>
<td><a href="http://example.org/ws">http://example.org/ws</a> rdf#type wsml#webService</td>
</tr>
</tbody>
</table>

### The second RDF graph for the web service description:

- http://example.org/ws rdf#type wsml#webService
- http://example.org/ws wsml#variant
- http://www.wsmo.org/wsml/wsml-syntax/wsml-flight
- http://example.org/ws wsml#useCapability
- http://example.org/ws wsml#useInterface
- http://example.org/ws wsml#useInterface http://example.org/i1
- http://example.org/ws wsml#useInterface http://example.org.net/i2
11 Mapping to OWL

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

11.1. Mapping WSML–Core to OWL DL

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

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

- Replace missing ontology identifier by the locator of the file containing the specification.
- Replace missing identifiers with unnumbered anonymous identifiers (i.e., for concepts, relations, instances, relation instances and axioms)
- Replace all unnumbered anonymous identifier by http://www.wsmo.org/wsml/wsml-syntax#anonymousID
- Replace idlists with single ids (in the case of ofType, impliesType, hasValue, subConceptOf and subRelationOf).

E.g., "hasAncestor hasValue [Peter, Paul]" is substituted by "hasAncestor hasValue Peter" and "hasAncestor hasValue Paul".
- All sQNames in the syntax are replaced with full IRI, according to the rules defined in Section 2.2
- Specifically within logical expression the following pre–processing steps are applied:
  - Replacing right implication with left implication (lexpr implies rexpr. := rexpr impliedBy lexpr).
  - Rewriting data term shortcuts ( "string" := _string("string"); integer := _integer("integer"); decimal := _decimal("decimal").
  - Rewriting all WSML datatype constructors according to Table C.1 to their corresponding XML schema datatype, e.g., _string to http://www.w3.org/2001/XMLSchema#string

Table 11.1 contains the mapping between the WSML Core and OWL DL abstract syntax through the mapping function $\tau$. In the table, underlined words refer to productions rules in the WSML grammar (see Appendix A) and boldfaced words refer to keywords in the WSML language. $X$ and $Y$ are meta–variables and are replaced with actual identifiers or variables during the translation itself. Note that in the table we use the familiar sQName macro mechanism (see also Section 2.1.2) to abbreviate IRI. The prefix 'wsml' stands for 'http://www.wsmo.org/wsml/wsml–syntax#', 'rdf' stands for 'http://www.w3.org/1999/02/22–rdf–syntax–ns#', 'rdfs' stands for 'http://www.w3.org/2000/01/rdf–schema#', 'dc' stands for 'http://purl.org/dc/elements/1.1#', 'xsd' stands for 'http://www.w3.org/2001/XMLSchema#', and 'owl' stands for 'http://www.w3.org/2002/07/owl#.'
<table>
<thead>
<tr>
<th><strong>WSML–Core conceptual syntax</strong></th>
<th><strong>OWL DL Abstract syntax</strong></th>
<th><strong>Remarks</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ ontology A header_1 ... header_n ontology_element_1 ... ontology_element_n }</code></td>
<td><code>Ontology(id </code>&lt;header_1&gt;<code>) ... </code>&lt;header_n&gt;<code>) </code>&lt;ontology_element_1<code>) ... </code>&lt;ontology_element_n`)</td>
<td>For non–functional properties on the ontology level &quot;Annotation&quot; instead of &quot;annotation&quot; has to be written.</td>
</tr>
<tr>
<td><code>{nonFunctionalProperties id_hasValue value ... id_hasValue value endNonFunctionalProperties }</code></td>
<td><code>annotation(id </code>&lt;value_1<code>) ... </code>&lt;value_n`)</td>
<td></td>
</tr>
<tr>
<td><code>{importsOntology idlist }</code></td>
<td><code>Annotation(owl#import </code>&lt;id_1<code>) ... </code>&lt;id_n`)</td>
<td>An <code>idlist</code> can consist of n full IRIs (this remark holds for all idlists in this table).</td>
</tr>
<tr>
<td><code>{usesMediator idlist }</code></td>
<td><code>Annotation(wsml#usesMediator </code>&lt;id_1<code>) ... </code>&lt;id_n`)</td>
<td>OWL does not have the concept of a mediator. Therefore, the translation uses the <code>wsml#usesMediator</code> annotation.</td>
</tr>
<tr>
<td><code>{concept id superconcept nfp att_id_1 att_type_1 range_id_1 ... att_id_n att_type_n range_id_n }</code></td>
<td><code>Class(id </code>&lt;nfp<code>) </code>&lt;superconcept<code>) </code>&lt;restriction (att_id_1 allValuesFrom range_id_1) ... <code>&lt;restriction (att_id_n allValuesFrom range_id_n) </code>[ObjectProperty</td>
<td>DatatypeProperty] (att_id_1) ... `[ObjectProperty</td>
</tr>
<tr>
<td><code>{subConceptOf idlist }</code></td>
<td><code>id_1 ... id_n</code></td>
<td></td>
</tr>
<tr>
<td><code>{relation id arity paramtyping superrelation nfp}</code></td>
<td>`[ObjectProperty</td>
<td>DatatypeProperty] (id <code>&lt;nfp</code>) <code>&lt;superrelation</code>) <code>&lt;paramtyping</code>)</td>
</tr>
<tr>
<td><code>{subRelationOf idlist }</code></td>
<td><code>super(id_1) ... super(id_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>{att_type domain_id_1 att_type range_id_1}</code></td>
<td><code>domain(domain_id_1 range(range_id_1)</code></td>
<td></td>
</tr>
<tr>
<td><code>{instance id memberof nfp att_id_1 hasValue value ... att_id_n hasValue value}</code></td>
<td><code>Individual(id </code>&lt;nfp<code>) </code>&lt;memberof<code>) </code>&lt;value (att_id_1 <code>&lt;value_1</code>) ... <code>&lt;value_n</code>)</td>
<td></td>
</tr>
<tr>
<td><code>{memberof idlist }</code></td>
<td><code>type(id_1) ... type(id_n)</code></td>
<td></td>
</tr>
<tr>
<td><code>{datatype_id(x_1,...,x_n)}</code></td>
<td><code>serialize</code>(datatype_id(x_1,...,x_n))<code>^^</code></td>
<td>parses WSML datatypes to XML Schema datatypes, according to Table C.1 in Appendix C.</td>
</tr>
</tbody>
</table>

Table 11.1: Mapping between WSML–Core and OWL DL abstract syntax
### 11.2. Partial Mapping from OWL DL to WSML-Core

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

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

In order to translate class axioms, the translation function τ takes a WSML logical expression from the translation of the left–hand side, τ₁ths of a subclass relation (or partial/class descriptions, respectively) as the first argument, the right hand side of a subclass relationship as second argument, and a variable as third argument. This variable is used for relating classes through properties from the
allValuesFrom and someValuesFrom restrictions. Whenever we pass such a value restriction during the translation, a new variable has to be introduced, i.e., \( x_{\text{new}} \) stands for a freshly introduced variable in every translation step.

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

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

<table>
<thead>
<tr>
<th>OWL DL Abstract syntax</th>
<th>WSML–Core syntax</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class Axioms</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class(( A ) partial ( C_1 \ldots C_n ))</td>
<td>concept ( A ) nonFunctionalProperties dc#relation hasValue AxiomOfA endNonFunctionalProperties</td>
<td></td>
</tr>
<tr>
<td>axiom AxiomOfA definedBy ( \tau (?x_{\text{new}} \text{memberOf} A, C_i, ?x_{\text{new}}) \ldots \tau (?x_{\text{new}} \text{memberOf} A, C_n, ?x_{\text{new}}) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If ( C_i ) is a named class the axiom definition for this ( i ) can be omitted and only conceptual syntax can be used by including ( C_i ) in the list of super concepts of ( A ): concept ( A ) subConceptOf ( { C } )</td>
<td></td>
</tr>
<tr>
<td>Class(( A ) complete ( C_1 \ldots C_n ))</td>
<td>concept ( A ) nonFunctionalProperties dc#relation hasValue AxiomOfA endNonFunctionalProperties</td>
<td></td>
</tr>
<tr>
<td>axiom AxiomOfA definedBy ( \tau (?x_{\text{new}} \text{memberOf} A, C_1, ?x_{\text{new}}) \ldots \tau (?x_{\text{new}} \text{memberOf} A, C_n, ?x_{\text{new}}) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \tau )_lhs stands for a transformation function for left-hand side descriptions.</td>
<td></td>
</tr>
<tr>
<td>EquivalentClasses(( C_1 \ldots C_n ))</td>
<td>axiom _# definedBy ( \tau )_lhs(( C_i, ?x_{\text{new}} ), ( C_j, ?x_{\text{new}} ))</td>
<td>Conjunctively for all ( i \neq j )</td>
</tr>
<tr>
<td>SubClassOf(( C_1, C_2 ))</td>
<td>axiom _# definedBy ( \tau )_lhs(( C_1, ?x_{\text{new}} ), ( C_2, ?x_{\text{new}} ))</td>
<td></td>
</tr>
</tbody>
</table>

**Mapping of left hand side descriptions**

- \( \tau \)\_lhs(\( A, X \)) \( X \text{memberOf} A \)
- \( \tau \)\_lhs(\( \text{intersectionOf}(C_1 \ldots C_n), X \)) \( \tau \)\_lhs(\( C_1, X \) \text{ and } \ldots \text{ and } \tau \)\_lhs(\( C_n, X \))
- \( \tau \)\_lhs(\( \text{unionOf}(C_1 \ldots C_n), X \)) \( \tau \)\_lhs(\( C_1, X \) \text{ or } \ldots \text{ or } \tau \)\_lhs(\( C_n, X \))
- \( \tau \)\_lhs(\( \text{restriction}(R \text{ someValuesFrom} C), X \)) \( X[R \text{ hasValue} ?x_{\text{new}}] \text{ and } \tau \)\_lhs(\( C, ?x_{\text{new}} \))
- \( \tau \)\_lhs(\( \text{restriction}(R \text{ minCardinality}(1)), X \)) \( X[R \text{ hasValue} ?x_{\text{new}}] \)

**Mapping of right hand side descriptions**

Table 11.2: Mapping between OWL DL abstract syntax and WSML–Core
<table>
<thead>
<tr>
<th>OWL DL Abstract syntax</th>
<th>WSML−Core syntax</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau(\text{WSMExpr}, \text{intersectionOf}(C_1 \ldots C_n), X) )</td>
<td>( \tau(\text{WSMExpr}, C_1, X) ) and ... and ( \tau(\text{WSMExpr}, C_n, X) )</td>
<td></td>
</tr>
<tr>
<td>( \tau(X \text{memberOf } A_1, A_2, X) )</td>
<td>( A_1 \text{ subConceptOf } A_2 )</td>
<td></td>
</tr>
<tr>
<td>( \tau(\text{WSMExpr}, A, X) )</td>
<td>( \text{WSMExpr implies } X \text{ memberOf } A )</td>
<td></td>
</tr>
<tr>
<td>( \tau(\text{WSMExpr}, \text{restriction}(R \text{ allValuesFrom } C), X) )</td>
<td>( \tau(\text{WSMExpr and } X[R \text{ hasValue } ?x_{\text{new}}], C, ?x_{\text{new}}) )</td>
<td></td>
</tr>
</tbody>
</table>

### Property Axioms

ObjectProperty(\( R \) [super(\( R_1 \) ... super(\( R_n \))])

- relation \( R/2 \)
- subRelationOf \( \{ R_1, \ldots, R_n \} \)
- nonFunctionalProperties
- df#relation hasValue AxiomOfR
dc#relation hasValue

endNonFunctionalProperties

axiom AxiomOfR

definedBy

[domain(\( C_j \) ... domain(\( C_n \))]

[range(\( D_j \) ... range(\( D_n \))]

[inverseOf(R)]

[Symmetric]

[Transitive]

\( ?x[R \text{ hasValue } ?y], C_j, ?x \) and ... and \( ?x[R \text{ hasValue } ?y], C_n, ?x \).

\( ?x[R \text{ hasValue } ?y], D_j, ?x \) and ... and \( ?x[R \text{ hasValue } ?y], D_n, ?x \).

\( ?x[R \text{ hasValue } ?y] \text{ implies } ?y[R' \text{ hasValue } ?x] \) and \( ?x[R' \text{ hasValue } ?y] \text{ implies } ?y[R \text{ hasValue } ?x] \).

\( ?x[R \text{ hasValue } ?y] \text{ implies } ?y[R \text{ hasValue } ?x] \).

\( ?x[R \text{ hasValue } ?y] \text{ implies } ?y[R \text{ hasValue } ?z] \).

\( ?x[R \text{ hasValue } ?y] \text{ implies } ?y[R \text{ hasValue } ?z] \).

SubProperty(\( R_1, R_2 \))

relation \( R_1/2 \) subRelationOf \( R_2 \)

EquivalentProperties(\( R_1 \ldots R_n \))

\( ?x[R_i \text{ hasValue } ?y] \text{ implies } ?x[R_j \text{ hasValue } ?y] \)

Conjunctively for all \( i \neq j \)

### Individuals

Individual(\( I \) [type(\( C_j \) ... type(\( C_n \))])

[range(\( D_j \) ... range(\( D_n \))]

[InverseOf(\( R \))]

[Symmetric]

[Transitive]

\( ?x[R_i \text{ hasValue } ?y] \text{ implies } ?x[R_j \text{ hasValue } ?y] \text{ implies } ?x[R_k \text{ hasValue } ?y] \).

\( ?x[R_i \text{ hasValue } ?y] \text{ implies } ?x[R_j \text{ hasValue } ?y] \).

\( ?x[R_i \text{ hasValue } ?y] \text{ and } ?x[R_j \text{ hasValue } ?y] \text{ and } ?x[R_k \text{ hasValue } ?y] \).

instance \( I \)

memberOf \( \{ C_1, \ldots, C_n \} \)

\( R_i \text{ hasValue } V_j \ldots \)

\( R_n \text{ hasValue } V_n \)

All Specifications of the Individual except the identifier are optional.
PART IV: FINALE

12. Implementation Efforts

WSMO4J (http://wsmo4j.sourceforge.net/) 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 (http://www.wsmo.org/2004/d2v1.0/). 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, 2005]. 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.
Appendix A. Human–Readable Syntax

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

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

A.1. BNF-Style Grammar

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

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

Helps

```
all                     = [ 0x0 .. 0xffff ]
escape_char             = ’\’
basechar                = [ 0x0041 .. 0x005A ] | [ 0x0061 .. 0x007A ]
ideographic            = [ 0x04E0 .. 0x09FA5 ] | [ 0x3007 | [ 0x3021 .. 0x3029 ]
letter                  = basechar | ideographic
digit                   = [ 0x0030 .. 0x0039 ]
combiningchar           = [ 0x0300 .. 0x0345 ] | [ 0x0360 .. 0x0361 ] | [ 0x0483 .. 0x0486 ]
extender                = [ 0x00B7 | [ 0x02DD | 0x02D1 | 0x0387 | 0x0640 | 0x0E46 | 0x0EC6 | 0x3005 | [ 0x3031 .. 0x3035 ] | [ 0x309D .. 0x309E] | [ 0x30FC .. 0x30FE ]
alphanum                 = digit | letter
hexdigit                = [ ’0’ .. ’9’ ] | [ ’A’ .. ’F’ ]
not_escaped_namechar    = letter | digit | ’_’ | combiningchar | extender
escaped_namechar        = ’*’ | ’+’ | not_escaped_namechar
namechar                = ( escape_char escaped_namechar ) | not_escaped_namechar
reserved                = ’?’ | ’?’ | ’#’ | ’@’ | ’&’ | ’?’ | ’$’ | ’,’
mark                    = ’*’ | ’?’ | ’_|’ | ’^’ | ’~’ | ’”’ | ’’ | ’?’ |
escaped                 = %’ hexdigit hexdigit
unreserved              = letter | digit | mark
scheme                  = letter ( letter | digit | ’<’ | ’<’ | ’<’ )
port                    = digit
idomainlabel            = alphanum ( ( alphanum | ’.’ )+ alphanum )?
dec_octet               = digit | ( [ 0x31 .. 0x39 ] digit ) | ( ’1’ digit digit ) | ( ’2’ [ 0x30 .. 0x34 ] digit ) | ( ’25’ [ 0x30 .. 0x35 ] )
ipv4address             = dec_octet ’.’ dec_octet ’.’ dec_octet ’.’ dec_octet
h4                      = hexdigit hexdigit hexdigit hexdigit
is32                    = ( h4 ’?’ h4 ) | ipv4address
```
Tokens

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_blank</td>
<td>= blank</td>
</tr>
<tr>
<td>t_comment</td>
<td>= comment</td>
</tr>
<tr>
<td>comma</td>
<td>= ','</td>
</tr>
<tr>
<td>endpoint</td>
<td>= ' ' blank</td>
</tr>
<tr>
<td>lpar</td>
<td>= '('</td>
</tr>
</tbody>
</table>

76 of 101
t_precondition = 'precondition'
t_reflexive = 'reflexive'
t_relation = 'relation'
t_relation_instance = 'relationInstance'
t_sharedvariable = 'sharedVariables'
t_source = 'source'
t_subconcept = 'subConceptOf'
t_subrelation = 'subRelationOf'
t_symmetric = 'symmetric'
t_target = 'target'
t_transitive = 'transitive'
t_usemediator = 'usesMediator'
t_useservice = 'usesService'
t_webservice = 'webService'
t_wgmmediator = 'wgmMediator'
t_wsmlvariant = 'wsmlVariant'
t_wwmediator = 'wwMediator'

variable = qmark alphanumeric+
anonymous = '_' '#'
nb_anonymous = '_' '#' digit+
pos_integer = digit+
pos_decimal = digit '.' digit+
string = dquote string_content dquote
full_iri = luridel iri reference luridel
name = ( letter | '_' ) namechar'

Ignored Tokens

• t_blank
• t_comment

Productions

wsml = wsmlvariant? namespace? definition*  
wsmlvariant = t_wsmlvariant full_iri 
namespace = t_namespace prefixdefinitionlist 
prefixdefinitionlist = (defaultns) full_iri 
| (prefixdefinitionlist) lbrace prefixdefinition moreprefixdefinitions* rbrace 
prefixdefinition = (namespacedef) name full_iri 
| (default) full_iri 
moreprefixdefinitions = comma prefixdefinition 
definition = (goal) goal 
| (ontology) ontology 
| (webservice) webservice 
| (mediator) mediator 
header = (nfp) nfp 
| (usesmediator) usesmediator 
| (importsontology) importsontology 
usesmediator = t_usemediator idlist 
importsontology = t_importontology idlist
nfp = t_nfp attributevalue* t_endnfp
mediator = (oomediator) oomediator
   | (ggmediator) ggmediator
   | (wgmediator) wgmediator
   | (wwmediator) wwmediator
ggmediator = t_ggmediator id? header? sources? target? use_service?
wgmediator = t_wgmediator id? header? source? target? use_service?
wwmediator = t_wwmediator id? header? source? target? use_service?
use_service = t_useservice id
source = t_source id
msources = t_source lbrace id moreide? rbrace
sources = (single) source
   | (multiple) msources
target = t_target id
goal = t_goal id? header? capability? interfaces?
webservice = t_webservice id? header? capability? interfaces?
capability = t_capability id? header? sharedvardef? pre_post_ass_or_eff
sharedvardef = t_sharedvariable variablelist
pre_post_ass_or_eff = (precondition) t_precondition axiomdefinition
   | (postcondition) t_postcondition axiomdefinition
   | (assumption) t_assumption axiomdefinition
   | (effect) t_effect axiomdefinition
interfaces = (single) interface
   | (multiple) minterfaces
minterfaces = t_interface lbrace id moreide? rbrace
interface = t_interface id? header? choreography? orchestration?
choreography = t_choreography id
orchestration = t_orchestration id
ontology = t_ontology id? header? ontology_element?
ontology_element = (concept) concept
   | (instance) instance
   | (relation) relation
   | (relationinstance) relationinstance
   | (axiom) axiom
concept = t_concept id superconcept? nfp? attribute*
superconcept = t_subconcept idlist
att_type = (open_world) t_ofype
   | (closed_world) t_impliestype
attribute = id attributefeature* att_type cardinality? idlist nfp?
cardinality = lpar pos integer cardinality_number? rpar
cardinality_number = (finite_cardinality) pos_integer
   | (infinite_cardinality) star
attributefeature = (transitive) t_transitive
   | (symmetric) t_symmetric
   | (inverse) t_inverseof lpar id rpar
   | (reflexive) t_reflexive
instance = t_instance id? memberof? nfp? attributevalue*
memberof = t_memberof idlist
attributevalue = id t_hasvalue valuelist
relation = t_relation id arity? paramtyping? superrelation? nfp?
paramtyping = lpar paramtype moreparamtype rpar
paramtype = att_type idlist
moreparamtype = comma paramtype
superrelation = t_subrelation idlist
arity = div_op pos_integer
relationinstance = t_relation_instance [name]: id? [relation]: id lpar value morevalues* rpar rpar nfp?
axiom = t_axiom axiomdefinition
axiomdefinition = (use_axiom) id
| (nfp_axiom) id? nfp
| (defined_axiom) id? nfp? log_definition
log_definition = t_definedby log_exp+
log_exp = (lp_rule) [head]: expr t_implied_by lp [body]: expr endpoint
| (constraint) t_constraint expr endpoint
| (other_expression) expr endpoint
expr = (implication) expr imply_op disjunction
| (disjunction) disjunction
disjunction = (conjunction) conjunction
| disjunction t or conjunction
conjunction = (subexpr) subexpr
| conjunction t and subexpr
subexpr = (negated) t not subexpr
| (simple) simple
| (complex) lpar expr rpar
| (quantified) quantified
quantified = quantifier_key variablelist lpar expr rpar
simple = (molecule) molecule
| (comparison) comparison
| (atom) term
molecule = (concept_molecule_preferred) term attr_specification? cpt_op term list
| (concept_molecule_nonpreferred) term cpt_op term list attr_specification
| (attribute_molecule) term attr_specification
attr_specification = lbracket attr_rel_list rbracket
attr_rel_list = (attr_relation) attr_relation
| attr_rel_list comma attr_relation
attr_relation = (attr_def) term attr_def_op term list
| (attr_val) term t_hasvalue term list
comparison = [left]: term comp_op [right]: term
functionsymbol = (parametrized) id lpar terms? rpar
| (math) lpar arith_val rpar
arith_val = (addition) arith_val arith_op mult_val
| (semisimple1_addition) term arith_op mult_val
| (semisimple2_addition) arith_val arith_op term
mult_val = [a]: term arith_op [b]: term
  | (multiplication) mult_val mul_op term
math_op = {arith} arith_op
  | {mult} mul_op
arith_op = {add} add_op
  | {sub} sub_op
mul_op = {mul} star
  | {div} div_op
comp_op = {gt}          gt
  | {lt}          lt
  | {gte}         gte
  | {lte}         lte
  | {equal}       equal
  | {strong_equal} strong_equal
  | {unequal}     unequal
cpt_op = {memberof} t_memberof
  | {subconceptof} t_subconcept
quantifier_key = {forall} t_forall
  | {exists} t_exists
attr_def_op = {oftype} t_oftype
  | {impliestype} t_impliestype
imply_op = {implies} t_implies
  | {implied_by} t_implied_by
  | {equivalent} t_equivalent
  | {implied_by_LP} t_implied_by_LP
prefix = name hash
sqname = {any} prefix? name
  | {localkeyword} prefix anykeyword
anykeyword = {and} t_and
  | {or} t_or
  | {implies} t_implies
  | {implied_by} t_implied_by
  | {equivalent} t_equivalent
  | {implied_by_LP} t_implied_by_LP
  | {constraint} t_constraint
  | {not} t_not
  | {exists} t_exists
  | {forall} t_forall
  | {univfalse} t_univfalse
  | {univtrue} t_univtrue
  | {assumption} t_assumption
  | {axiom} t_axiom
  | {capability} t_capability
  | {choreography} t_choreography
  | {concept} t_concept
  | {definedby} t_definedby
  | {effect} t_effect
  | {endnfp} t_endnfp
  | {ggmediator} t_ggmediator
  | {goal} t_goal
{hasvalue}  t_hasvalue
{impliestype}  t_impliestype
{importontology}  t_importontology
{instance}  t_instance
{interface}  t_interface
{inverseof}  t_inverseof
{memberof}  t_memberof
{namespace}  t_namespace
{nfp}  t_nfp
{oftype}  t_oftype
{ontology}  t_ontology
{oomediato}  t_oomediato
{orchestration}  t_orchestration
{postcondition}  t_postcondition
{precondition}  t_precondition
{reflexive}  t_reflexive
{relation}  t_relation
{relation_instance}  t_relation_instance
{sharedvariable}  t_sharedvariable
{source}  t_source
{subconcept}  t_subconcept
{subrelation}  t_subrelation
{symmetric}  t_symmetric
{target}  t_target
{transitive}  t_transitive
{usemediato}  t_usemediato
{useservice}  t_useservice
{webservice}  t_webservice
{wgmmediato}  t_wgmmediato
{wamlivariant}  t_wamlivariant
{wwmediato}  t_wwmediato

iri    = (iri)  full iri
       | {sqname}  sqname

id     = (iri)  full id
       | {anonymous}  anonymous
       | {universal_truth}  t_univtrue
       | {universal_falsehood}  t_univfalse

idlist = (id)  id
       | {idlist}  brace id moreids* rbrace

moreids = comma  id

value  = (datatype)  functionsymbol
       | {term}  id
       | {numeric}  number
       | {string}  string

valuelist = {term}  value
       | {valuelist}  brace  value  morevalues*  rbrace

morevalues = comma  value

term   = (data)  value
       | {var}  variable
       | {nb_anonymous}  nb_anonymous
A.2. Example of the Human-Readable Syntax

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

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

/***************************************************************************/
* attributes.
* This Concept illustrates the use of different styles of
* attributes.
+
/* concept Human
   nonFunctionalProperties:
       dc@title hasValue "concept of a human being"
endNonFunctionalProperties
hasName ofType foaf#name
hasParent inverseOf(hasChild) impliesType Human
hasChild impliesType Human
hasAncestor transitive impliesType Human
hasWeight ofType _decimal
hasWeightInKG ofType _decimal
hasBirthdate ofType _date
hasBirthplace ofType _date
isMarriedTo symmetric impliesType (0 1) Human
hasCitizenship ofType _integer
isAlive ofType (1) boolean
nfp
dc@relation hasValue [isAlive]
endnfp

relation ageOfHuman (ofType Human, ofType _integer)
nfp
dc@relation hasValue [FunctionalDependencyAge]
endnfp

axiom IsAlive definedBy
   @if {isAlive hasValue [false]} :-
      naf @if {hasObit hasValue [true]} memberOf Human.
   @if {isAlive hasValue [true]}
      impliedBy
/***************************************************************************/
?x[hasObit hasValue ?obit] memberOf Human.

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

concept Man subConceptOf Human
def
doRelation hasValue ManDisjointWoman
endDef

concept Woman subConceptOf Human
def
doRelation hasValue ManDisjointWoman
endDef

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

/*
 * Refining a concept and restricting an existing attribute
 */
concept Parent subConceptOf Human
def
doDescription hasValue "Human with at least one child"
endDef

/*
 * Using an axiom to define class membership and an additional
 * axiom as constraint
 */
concept Child subConceptOf Human
def
doRelation hasValue {ChildDef, ValidChild}
endDef

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

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

/*
 * Defining complete subclasses by use of axioms
 */
concept Girl subConceptOf Woman
def
doRelation hasValue CompletenessOfChildren
endDef

concept Boy
def
doRelation hasValue {ABoy, CompletenessOfChildren}
endDef

/*
 * This axiom implies that Boy is a Man and a Child and every Man which
 * is also a Child is a Boy
 */
axiom ABoy
definedBy
?x memberOf Boy equivalent ?x memberOf Man and ?x memberOf Child.

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

instance Mary memberOf {Parent, Woman}
endInstance
hasName hasValue "Mary Smith"
hasBirthdate hasValue _date(1949,09,12)
hasChild hasValue { Paul, Susan }
instance Paul memberOf { Parent, Man }
  hasName hasValue "Paul Smith"
  hasBirthdate hasValue _date(1976,08,16)
  hasChild hasValue George
  hasCitizenship hasValue oo#de
instance Susan memberOf Woman
  hasName hasValue "Susan Jones"
  hasBirthdate hasValue _date(1976,08,16)

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

relationInstance ageOfHuman(George, 1)

* WEBSERVICE
**************************
webService _"http://example.org/Germany/BirthRegistration"
  nfp
dc#description hasValue "Birth registration service for Germany"  
dc#type hasValue _"http://www.wsmo.org/TR/d2/v1.2/#services"
  wsmi#version hasValue "$Revision: 1.9 $"
endnfp
usesMediator (_"http://example.org/ooMediator")
importsOntology (_"http://www.example.org/ontologies/example", _"http://www.wsmo.org/ontologies/location")
capability
  sharedVariables ?child
  precondition
    nonFunctionalProperties
    dc#description hasValue "The input has to be boy or a girl
    with birthdate in the past and be born in Germany."  
endNonFunctionalProperties
  definedBy
    ?child memberOf Child
    and ?child[hasBirthdate hasValue ?birthdate]
    and wsmi#dateLessThan(?birthdate, wsmi#currentDate())
    and ?child[hasBirthplace hasValue ?location]
    and ?location[locatedIn hasValue oo#de]
    or (?child[hasParent hasValue ?parent]
    and ?parent[hasCitizenship hasValue oo#de] ).
  assumption
    nonFunctionalProperties
    dc#description hasValue "The child is not dead"
endNonFunctionalProperties
  definedBy
    ?child memberOf Child
    and naf ?child[hasObit hasValue ?x].
  effect
    nonFunctionalProperties
    dc#description hasValue "After the registration the child
    is a German citizen"  
endNonFunctionalProperties
  definedBy
    ?child memberOf Child
    and ?child[hasCitizenship hasValue oo#de].
interface
  choreography _"http://example.org/tobedone"
  orchestration _"http://example.org/tobedone"

* GOAL
**************************
goal _"http://example.org/Germany/GetCitizenShip"
  nonFunctionalProperties
  dc#description hasValue "Goal of getting a citizenship within Germany"  
dc#type hasValue _"http://www.wsmo.org/TR/d2/v1.2/#goals"
  wsmi#version hasValue "$Revision: 1.9 $"
endNonFunctionalProperties
usesMediator (_"http://example.org/ooMediator")
importsOntology (_"http://www.example.org/ontologies/example", _"http://www.wsmo.org/ontologies/location")
capability
  sharedVariables ?human
  effect havingACitizenShip
    nonFunctionalProperties
This goal expresses the general desire of becoming a citizen of Germany.

Goal of getting a Registration for Paul's son George

decription hasValue "This goal expresses Paul's desire to register his son with the German birth registration board."
definedBy

importOntology (_http://example.org/ontologies/example",

webService bankTransaction
capability

precondition
definedBy

postcondition
definedBy

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').

In the general case the generic goal and the WS are known before a concrete request is made and can be statically linked, to avoid reasoning during the runtime of a particular request. The fact that the WS fulfills at least partially the goal is explicitly stated in the wgMediator.
Appendix B. Schemas for the XML Exchange Syntax

In the following sections we present the XML Schemas for the XML syntax of WSML, which was introduced in Chapter 9. The schemas are available online at http://www.wsmo.org/TR/d16/d16.1/v0.21/xml-syntax/wsml–xml-syntax.xsd.

This schema includes two module schemas:

- WSML identifiers (http://www.wsmo.org/TR/d16/d16.1/v0.21/xml–syntax/wsml–identifiers.xsd)
- Logical expressions of WSML (http://www.wsmo.org/TR/d16/d16.1/v0.21/xml–syntax/wsml–expr.xsd).

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

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

- XML syntax for WSML logical expressions: http://www.wsmo.org/TR/d16/d16.1/v0.21/xml–syntax/documentation/wsml–expr.xsd.html
Appendix C. Datatypes and Built−ins in WSML

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

Appendix C.1. WSML Datatypes

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

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

<table>
<thead>
<tr>
<th>WSML datatype constructor</th>
<th>XML Schema datatype</th>
<th>Description of the Datatype</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>_string</td>
<td>string</td>
<td>A finite−length sequence of Unicode characters, where each occurrence of the double quote &quot; is escaped using the backslash symbol: &quot; and the backslash is escaped using the backslash: \.</td>
<td>_string(&quot;any−character&quot;)</td>
</tr>
<tr>
<td>_decimal</td>
<td>decimal</td>
<td>That subset of natural numbers which can be represented with a finite sequence of decimal numerals.</td>
<td>_decimal(&quot;−?numeric+.numeric+&quot;)</td>
</tr>
<tr>
<td>_integer</td>
<td>integer</td>
<td>That subset of decimals which corresponds with natural numbers.</td>
<td>_integer(&quot;−?numeric+&quot;)</td>
</tr>
<tr>
<td>_float</td>
<td>float</td>
<td>_float(&quot;see XML Schema document&quot;)</td>
<td></td>
</tr>
<tr>
<td>_double</td>
<td>double</td>
<td>_double(&quot;see XML Schema document&quot;)</td>
<td></td>
</tr>
<tr>
<td>_iri</td>
<td>similar to anyURI</td>
<td>An IRI conforms to [Duerst &amp; Suignard, 2005]. Every URI is an IRI.</td>
<td>_iri(&quot;iri−according−to−rfc3987&quot;)</td>
</tr>
<tr>
<td>_qname</td>
<td>serialized QName</td>
<td>An sQName is a pair (namespace, localname), where the localname us concatenated to the namespace to form an IRI. An sQName is actually equivalent to the IRI which results from concatenating the namespace and the localname.</td>
<td>_qname(&quot;iri−according−to−rfc3987&quot;, &quot;localname&quot;)</td>
</tr>
<tr>
<td>_boolean</td>
<td>boolean</td>
<td>_boolean(&quot;true−or−false&quot;)</td>
<td></td>
</tr>
<tr>
<td>_duration</td>
<td>duration</td>
<td>_duration(year, month, day, hour, minute, second)</td>
<td></td>
</tr>
<tr>
<td>_dateTime</td>
<td>dateTime</td>
<td>_dateTime(year, month, day, hour, minute, second, timezone–hour, timezone–minute) _dateTime(year, month, day, hour, minute, second)</td>
<td></td>
</tr>
<tr>
<td>_time</td>
<td>time</td>
<td>_time(hour, minute, second, timezone–hour, timezone–minute) _time(hour, minute, second)</td>
<td></td>
</tr>
<tr>
<td>_date</td>
<td>date</td>
<td>_date(year, month, day, timezone–hour, timezone–minute) _date(year, month, day)</td>
<td></td>
</tr>
</tbody>
</table>

Table C.1: WSML Datatypes
<table>
<thead>
<tr>
<th>WSML datatype constructor</th>
<th>XML Schema datatype</th>
<th>Description of the Datatype</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>_gyearmonth</td>
<td>gYearMonth</td>
<td>_gyearmonth(year, month)</td>
<td></td>
</tr>
<tr>
<td>_gyear</td>
<td>gYear</td>
<td>_gyear(year)</td>
<td></td>
</tr>
<tr>
<td>_gmonthday</td>
<td>gMonthDay</td>
<td>_gmonthday(month, day)</td>
<td></td>
</tr>
<tr>
<td>_gday</td>
<td>gDay</td>
<td>_gday(day)</td>
<td></td>
</tr>
<tr>
<td>_gmonth</td>
<td>gMonth</td>
<td>_gmonth(month)</td>
<td></td>
</tr>
<tr>
<td>_hexbinary</td>
<td>hexBinary</td>
<td>_hexbinary(hexadecimal–encoding)</td>
<td></td>
</tr>
<tr>
<td>_base64binary</td>
<td>base64Binary</td>
<td>_base64binary(hexadecimal–encoding)</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>WSML datatype constructor</th>
<th>Shortcut syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>_string</td>
<td>&quot;any–character&quot;**</td>
<td>&quot;John Smith&quot;</td>
</tr>
<tr>
<td>_decimal</td>
<td>'-?numeric+.numeric+</td>
<td>4.2, 42.0</td>
</tr>
<tr>
<td>_integer</td>
<td>'-?numeric+</td>
<td>42, −4</td>
</tr>
<tr>
<td>_sqname</td>
<td>alphanumeric+&quot;#&quot;alphabetic+</td>
<td>wsml#concept, xsd#integer</td>
</tr>
</tbody>
</table>

Table C.2: WSML Datatype shortcut syntax

Appendix C.2. WSML Built-in Predicates

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

The current list is only based on the built-in support in the WSML language through the use of special symbols. A translation of the built-in symbols to datatype predicates is given in the next section. The symbol 'range' signifies the range of the function. Functions in XQuery have a defined range, whereas predicates only have a domain. Therefore, the first argument of a WSML datatype predicate which represents a function represents the range of the function. Comparators in XQuery are functions, which return a boolean value. These comparators are directly translated to predicates. If the XQuery function returns 'true', the arguments of the predicate are in the extension of the predicate. See Table C.3 for the complete list. In case the built-in predicate has an equivalent function in XQuery, this is indicated in the table. The types 'wsml#equal', 'wsml#inequal' and 'wsml#strongEqual' refer to the unification operator, inequality and user-defined inequality, respectively.
<table>
<thead>
<tr>
<th>WSML built-in predicate</th>
<th>XQuery function</th>
<th>Type (A)</th>
<th>Type (B)</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsml#equal</td>
<td></td>
<td>abstract</td>
<td>abstract</td>
<td></td>
</tr>
<tr>
<td>wsml#inequal</td>
<td></td>
<td>abstract</td>
<td>abstract</td>
<td></td>
</tr>
<tr>
<td>wsml#StrongEqual</td>
<td></td>
<td>abstract</td>
<td>abstract</td>
<td></td>
</tr>
<tr>
<td>wsml#numericEqual(A,B)</td>
<td>op#numeric−equal(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td></td>
</tr>
<tr>
<td>wsml#numericGreaterThan(A,B)</td>
<td>op#numeric−greater-than(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td></td>
</tr>
<tr>
<td>wsml#numericLessThan(A,B)</td>
<td>op#numeric−less-than(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td></td>
</tr>
<tr>
<td>wsml#stringEqual(A,B)</td>
<td>op#numeric−equal(fn:compare(A, B), 1)</td>
<td>xsd#string</td>
<td>xsd#string</td>
<td></td>
</tr>
<tr>
<td>wsml#numericAdd(range,A,B)</td>
<td>op#numeric−add(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml#numericSubtract(range,A,B)</td>
<td>op#numeric−subtract(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml#numericMultiply(range,A,B)</td>
<td>op#numeric−multiply(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
<tr>
<td>wsml#numericDivide(range,A,B)</td>
<td>op#numeric−divide(A,B)</td>
<td>numeric</td>
<td>numeric</td>
<td>numeric</td>
</tr>
</tbody>
</table>

Table C.3: WSML Built-in Predicates

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

- wsml#equal, wsml#inequal and wsml#StrongEqual are not required for WSML-Core
- wsml#StrongEqual is not required for WSML-Flight and WSML-Rule

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

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

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

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type (A)</th>
<th>Type (B)</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B</td>
<td>abstract</td>
<td>abstract</td>
<td>wsml#equal(A,B)</td>
</tr>
<tr>
<td>A != B</td>
<td>abstract</td>
<td>abstract</td>
<td>wsml#inequal(A,B)</td>
</tr>
<tr>
<td>A &gt;= B</td>
<td>abstract</td>
<td>abstract</td>
<td>wsml#StrongEqual(A,B)</td>
</tr>
<tr>
<td>A = B</td>
<td>string</td>
<td>string</td>
<td>wsml#stringEqual(A,B)</td>
</tr>
<tr>
<td>A != B</td>
<td>xsd#string</td>
<td>xsd#string</td>
<td>wsml#stringInequal(A,B)</td>
</tr>
<tr>
<td>A = B</td>
<td>numeric</td>
<td>numeric</td>
<td>wsml#numericEqual(A,B)</td>
</tr>
<tr>
<td>A != B</td>
<td>numeric</td>
<td>numeric</td>
<td>wsml#numericInequal(A,B)</td>
</tr>
<tr>
<td>A &lt; B</td>
<td>numeric</td>
<td>numeric</td>
<td>wsml#lessThan(A,B)</td>
</tr>
<tr>
<td>A &lt;= B</td>
<td>numeric</td>
<td>numeric</td>
<td>wsml#lessEqual(A,B)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>numeric</td>
<td>numeric</td>
<td>wsml#greaterThan(A,B)</td>
</tr>
<tr>
<td>A &gt;= B</td>
<td>numeric</td>
<td>numeric</td>
<td>wsml#greaterEqual(A,B))</td>
</tr>
</tbody>
</table>

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

We list the built-in functions and their translation to datatype predicates in Table C.5. In the table, ?x1 represents a unique newly introduced variable, which stands for the range of the function.
Function symbols in WSML are not as straightforward to translate to datatype predicates as are relations. However, if we see the predicate as a function, which has the range as its first argument, we can introduce a new variable for the return value of the function and replace an occurrence of the function symbol with the newly introduced variable and append the newly introduced predicate to the conjunction of which the top-level predicate is part.

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

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

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

\[ ?w = ?x + ?y + ?z \]

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

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

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

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

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

\[ wsml#numericEqual(?w, ?x2) \text{ and } wsml#numericAdd(?x1, ?x, ?y) \text{ and } wsml#numericAdd(?x2, ?x1, ?z) \]
Appendix D. WSML Keywords

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

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

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

We list all keywords of the WSML conceptual syntax in Table D.1.
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Section</th>
<th>Core</th>
<th>Flight</th>
<th>Rule</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsmiVariant</td>
<td>2.2.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>namespace</td>
<td>2.2.2</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>nonFunctionalProperties</td>
<td>2.2.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>endNonFunctionalProperties</td>
<td>2.2.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>nfp</td>
<td>2.2.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>endnfp</td>
<td>2.2.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>importsOntology</td>
<td>2.2.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>usesMediator</td>
<td>2.2.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ontology</td>
<td>2.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>concept</td>
<td>2.3.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>subConceptOf</td>
<td>2.3.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ofType</td>
<td>2.3.1.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>impliesType</td>
<td>2.3.1.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>transitive</td>
<td>2.3.1.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>symmetric</td>
<td>2.3.1.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>inverseOf</td>
<td>2.3.1.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>reflexive</td>
<td>2.3.1.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>relation</td>
<td>2.3.2</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>subRelationOf</td>
<td>2.3.2</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>instance</td>
<td>2.3.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>memberOf</td>
<td>2.3.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>hasValue</td>
<td>2.3.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>relationInstance</td>
<td>2.3.3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>axiom</td>
<td>2.3.4</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>definedBy</td>
<td>2.3.4</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>capability</td>
<td>2.4.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>sharedVariables</td>
<td>2.4.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>precondition</td>
<td>2.4.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>assumption</td>
<td>2.4.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>postcondition</td>
<td>2.4.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>effect</td>
<td>2.4.1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>interface</td>
<td>2.4.2</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>choreography</td>
<td>2.4.2</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>orchestration</td>
<td>2.4.2</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>goal</td>
<td>2.5</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Table D.2 lists the keywords allowed in logical expressions. The complete logical expression syntax is defined in Section 2.8. Besides the keywords in this list, WSML also allows for the use of a number of infix operators for built-in predicates, see also Appendix C.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>WSML−Core</th>
<th>WSML−Flight</th>
<th>WSML−Rule</th>
<th>WSML−Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>false</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>memberOf</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>hasValue</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>subConceptOf</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ofType</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>impliesType</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>and</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>or</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>implies</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>impliedBy</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>equivalent</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>neg</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>naf</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>forall</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>exists</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
</tbody>
</table>

[TODO: Check keyword/variant combos] Table D.2: WSML logical expression keywords
Appendix E. Relation to WSMO Conceptual Model

WSML aims at providing a complete language to describe all elements in the WSMO conceptual model as defined in [Roman et al. 2005]. However, although most elements in WSMO have direct correspondences in WSML language constructs there are slight differences due to the fact that WSML is in principle a logical description language rather than a conceptual model.

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

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

```
relation ageOfHuman2 (oType Human, oType xsd#integer)
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>oType</td>
<td>Human</td>
</tr>
<tr>
<td>ofType</td>
<td>xsd#integer</td>
</tr>
<tr>
<td>oPredicate</td>
<td>hasValue</td>
</tr>
<tr>
<td>oDomain</td>
<td>AgeOfHuman</td>
</tr>
<tr>
<td>oInverseDomain</td>
<td>FunctionalDependencyAge</td>
</tr>
</tbody>
</table>

axiom AgeOfHuman definedBy
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>oPredicate</td>
<td>hasValue</td>
</tr>
<tr>
<td>oDomain</td>
<td>AgeOfHuman</td>
</tr>
<tr>
<td>oInverseDomain</td>
<td>oPredicate value</td>
</tr>
<tr>
<td>oArgument1</td>
<td>x</td>
</tr>
<tr>
<td>oArgument2</td>
<td>y</td>
</tr>
<tr>
<td>oType</td>
<td>functionDependencyAge</td>
</tr>
<tr>
<td>oInverseType</td>
<td>xsd#integer</td>
</tr>
<tr>
<td>oPredicate</td>
<td>hasValue</td>
</tr>
<tr>
<td>oDomain</td>
<td>AgeOfHuman</td>
</tr>
<tr>
<td>oInverseDomain</td>
<td>oPredicate value</td>
</tr>
<tr>
<td>oArgument1</td>
<td>x</td>
</tr>
<tr>
<td>oArgument2</td>
<td>y</td>
</tr>
<tr>
<td>oType</td>
<td>functionDependencyAge</td>
</tr>
<tr>
<td>oInverseType</td>
<td>xsd#integer</td>
</tr>
<tr>
<td>oPredicate</td>
<td>hasValue</td>
</tr>
<tr>
<td>oDomain</td>
<td>AgeOfHuman</td>
</tr>
<tr>
<td>oInverseDomain</td>
<td>oPredicate value</td>
</tr>
<tr>
<td>oArgument1</td>
<td>x</td>
</tr>
<tr>
<td>oArgument2</td>
<td>y</td>
</tr>
</tbody>
</table>

```

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

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

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

Choreography and Orchestration Interfaces. The language for defining complex interfaces of Web services is not yet defined within WSML. however, we expect it to be based on the initial drafts in
[Scicluna et al., 2005] where we aim at refining the language and semantics used there to a more user–friendly level.
Appendix F. Changelog

Compared with the previous version of this document (v0.2 – 2005–03–20), the following changes have been made:

1. **Appendix C** now defines the URIs for built-ins for equality:
   - Unification operator: wsml#equal (shortcut syntax: =)
   - Inequality: wsml#inequal (shortcut syntax: !=)
   - User-defined equality operator: wsml#strongEqual (shortcut syntax: :=:)
   - The shortcut symbol for strong equality has been added to the grammar in Appendix A.
2. The **erratum to D16.1v0.2** has been implemented.
3. In the XML Schema for WSML identifiers, the XML Schema type xs:anyURI is now used as a basis for wsmlIRI, rather than xs:string.
4. The **definition of WSML-Core logical expressions** now explains what 'lhs' and 'rhs' stand for.
5. The **definition of atomic formulae in the logical expression syntax** defines molecules.
6. The **WSML-Rule logical expression syntax** has been generalized to allow both universal and existential quantification in the body of a rule. The **definition of the semantics** simply uses Lloyd-Topor transformations to eliminate these quantifications.
7. **WSML-Rule** now allows unstratified negation under the Well-Founded Semantics.
8. A note was added about the use of invalid identifiers at the end of the section on identifiers.
9. Nesting of classical implication in the head of a WSML Flight/Rule rule has been disallowed.
10. The **WSML-Flight and WSML-Rule** semantics now define transformation rules for eliminating negation on non-atomic formulae.
11. The examples of WSML-Core, WSML-Flight and WSML-Rule logical expressions have been updated.
12. The syntax of WSML-DL logical expressions and the WSML-DL semantics have been added.
References


http://www.wsmo.org/2004/d4/d4.1/v0.1/


Acknowledgements

The work is funded by the European Commission under the projects DIP, Knowledge Web, InfraWebs, SEKT, SWWS, ASG and Esperonto; by Science Foundation Ireland under the DERI–Lion project; by the FIT–IT (Forschung, Innovation, Technologie – Informationstechnologie) under the projects RW² and TSC.

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

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

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

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