WP 1: Ontology Reasoning and Querying

D1.7

Semantics

Jos de Bruijn
Axel Polleres
Holger Lausen
Livia Predoiu

June 6th, 2005
EXECUTIVE SUMMARY

In this document, we provide a mathematical foundation of the semantics of the Web Service Modelling Language WSML, which provides a formal syntax and semantics for the Web Service Modelling Ontology WSMO. WSML is based on different logical formalisms, namely, Description Logics, First-Order Logic and Logic Programming, which are useful for the modelling of Semantic Web services.

The framework of 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.

**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.

In this document, we focus on the semantics of the logical expressions within the WSML languages. The syntax has already been introduced in the DIP deliverable D2.7. At the time being, the syntax of all five WSML languages has been defined. However, the semantics has not been defined yet for all WSML languages, namely the semantical specification of the WSML variants WSML-DL and WSML-Full are missing. We intend to provide a mathematical foundation of the semantics of WSML-DL soon.
WSML-Full semantics will not be provided because we disadvice the usage of it within the DIP project. Because WSML-Full unifies the other WSML-variants, it is a very expressive language which aims at extending First-Order Logic with nonmonotonic features. For the purposes of describing Web Services in a manner that makes it feasible to deduce new information or to reason, respectively, in realistic time, it is not appropriate. The high expressivity brings along highly intractable reasoning algorithms. Therefore, the usage of WSML-Full in real-world applications is not recommended.

Because the semantics of capability descriptions is not entirely clear at the moment, we only define the semantics of ontology definitions within this document.

The deliverable is intended for everybody who is interested in and concerned with the formal description of ontologies as well as web services. This applies in particular to all partners in DIP that are involved in the development of ontology related tools.

Disclaimer: The DIP Consortium is proprietary. There is no warranty for the accuracy or completeness of the information, text, graphics, links or other items contained within this material. This document represents the common view of the consortium and does not necessarily reflect the view of the individual partners.
Ontology Representation Language

Document Information

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>FP6 – 507483</th>
<th>Acronym</th>
<th>DIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full title</td>
<td>Data, Information, and Process Integration with Semantic Web Services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project URL</td>
<td><a href="http://dip.semanticweb.org">http://dip.semanticweb.org</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document URL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EU Project officer</td>
<td>Kai Tullius</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>1.7</th>
<th>Title</th>
<th>Ontology Representation Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work package Number</td>
<td>1</td>
<td>Title</td>
<td>Ontology Management</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date of delivery</th>
<th>Contractual M 18</th>
<th>Actual 30-Jun-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>version. 1.1 final</td>
<td>final ✓</td>
</tr>
<tr>
<td>Nature</td>
<td>Prototype ☒ Report ☒ Dissemination ☒</td>
<td></td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>Public ☒ Consortium ☒</td>
<td></td>
</tr>
</tbody>
</table>

Authors (Partner) Jos de Bruijn (UIBK), Axel Polleres (UIBK), Holger Lausen (UIBK), Livia Predoiu (UIBK)

<table>
<thead>
<tr>
<th>Responsible Author</th>
<th>Livia Predoiu</th>
<th>Email</th>
<th><a href="mailto:livia.predoiu@deri.org">livia.predoiu@deri.org</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Partner</td>
<td>UIBK</td>
<td>Phone</td>
<td>+43 512 507 6111</td>
</tr>
</tbody>
</table>

Abstract (for dissemination) We present the semantics of WSML, which is thoroughly described in the DIP deliverable D2.7.

Keywords Semantics, WSML

Version Log

<table>
<thead>
<tr>
<th>Issue Date</th>
<th>Rev No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-06-05</td>
<td>0.1</td>
<td>Livia Predoiu</td>
<td>Set up initial document structure</td>
</tr>
<tr>
<td>06-06-05</td>
<td>1.0</td>
<td>Livia Predoiu</td>
<td>Finished document for review</td>
</tr>
</tbody>
</table>

Deliverable 2.7 iii 16/07/20045
## Project Consortium Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| National University of Ireland Galway | NUIG | Prof. Dr. Christoph Bussler  
Digital Enterprise Research Institute (DERI)  
National University of Ireland, Galway  
Galway  
Ireland  
Email: chris.bussler@deri.org  
Tel: +353 91 512460 |
| Fundacion De La Innovacion.Bankinter | Bankinter | Monica Martinez Montes  
Fundacion de la Innovation. Bankinter  
Paseo Castellana, 29  
28046 Madrid,  
Spain |
| Berlecon Research GmbH | Berlecon | Dr. Thorsten Wichmann  
Berlecon Research GmbH  
Oranienburger Str. 32  
10117 Berlin,  
Germany |
| British Telecommunications Plc. | BT | Dr. John Davies  
BT Exact (Orion Floor 5 pp12)  
Adastral Park Martlesham  
Ipswich IP5 3RE,  
United Kingdom |
| Swiss Federal Institute of Technology, Lausanne | EPFL | Prof. Karl Aberer  
Distributed Information Systems Laboratory  
École Polytechnique Fédérale de Lausanne  
Bât. PSE-A  
1015 Lausanne, Switzerland  
Email: Karl.Aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council | Essex | Mary Rowlett,  
Essex County Council  
PO Box 11, County Hall, Duke Street  
Chelmsford, Essex, CM1 1LX  
United Kingdom.  
Email: mary@essexcc.gov.uk  
Tel: +44 (0)1245 436524 |
| Forschungszentrum Informatik | FZI | Andreas Abecker  
Forschungszentrum Informatik  
Haid-und-Neu Strasse 10-14  
76131 Karlsruhe  
Germany |

Deliverable 2.7 iv 16/07/20045
<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| Institut für Informatik, Leopold-Franzens Universität Innsbruck | UIBK | Prof. Dieter Fensel  
Institute of computer science  
University of Innsbruck  
Technikerstr. 25  
A-6020 Innsbruck, Austria  
Email: dieter.fensel@deri.org  
Tel: +43 512 5076485 |
| ILOG SA | ILOG | Christian de Sainte Marie  
9 Rue de Verdun, 94253  
Gentilly, France  
Email: csma@ilog.fr  
Tel: +33 1 49082981 |
| inubit AG | Inubit | Torsten Schmale  
inubit AG  
Lützowstraße 105-106  
D-10785 Berlin  
Germany  
Email: ts@inubit.com  
Tel: +49 30726112 0 |
| Intelligent Software Components, S.A. | iSOCO | Dr. V. Richard Benjamins, Director R&D  
Intelligent Software Components, S.A.  
Pedro de Valdivia 10  
28006 Madrid, Spain  
Email: rbenjamins@isoco.com  
Tel. +34 913 349 797 |
| NIWA WEB Solutions | NIWA | Alexander Wahler  
NIWA WEB Solutions  
Niederacher & Wahler OEG  
Kirchengasse 13/1a  
A-1070 Wien  
Email: wahler@niwa.at  
Tel:+43(0)1 3195843-11 |
| The Open University | OU | Dr. John Domingue  
Knowledge Media Institute  
The Open University, Walton Hall  
Milton Keynes, MK7 6AA  
United Kingdom  
Email: j.b.domingue@open.ac.uk  
Tel.: +44 1908 655014 |
| SAP AG | SAP | Dr. Elmar Dörner  
SAP Research, CEC Karlsruhe  
SAP AG  
Vincenz-Priessnitz-Str. 1  
76131 Karlsruhe, Germany  
Email: elmar.dörner@sap.com  
Tel: +49 721 6902 31 |
<table>
<thead>
<tr>
<th>Company</th>
<th>Contact Information</th>
</tr>
</thead>
</table>
| Sirma AI Ltd.        | Atanas Kiryakov, Ontotext Lab, Sirma AI EAD  
Office Express IT Centre, 3rd Floor  
135 Tzarigradsko Chausse  
Sofia 1784, Bulgaria  
Email: atanas.kiryakov@sirma.bg  
Tel.: +359 2 9768 303 |
| Unicorn Solution Ltd.| Jeff Eisenberg,  
Unicorn Solutions Ltd,  
Malcha Technology Park 1  
Jerusalem 96951  
Israel  
Email: Jeff.Eisenberg@unicorn.com  
Tel.: +972 2 6491111 |
| Vrije Universiteit Brussel | Carlo Wouters,  
Starlab- VUB  
Vrije Universiteit Brussel  
Pleinlaan 2, G-10  
1050 Brussel, Belgium  
Email: carlo.wouters@vub.ac.be  
Tel.: +32 (0) 2 629 3719 |
LIST OF KEY WORDS/ABBREVIATIONS

Semantics of WSML


WSMO  – Web Service Modelling Ontology (see www.wsmo.org)

WSMX – Web Service Execution Environment (see www.wsmo.org/wsmx)
# Table of Contents

**Executive Summary** .................................................................................................................. I

**List of Key Words/Abbreviations** ......................................................................................... VII

**Table of Contents** ................................................................................................................... VIII

1 **Introduction** ........................................................................................................................... 1

2 **Syntax of Logical Expressions** ........................................................................................... 4
   2.1 WSML-Full .......................................................................................................................... 4
      2.1.1 Ontology Specifications in WSML-Full ................................................................. 4
      2.1.2 Capability and Interface Specification in WSML .................................................. 10
      2.1.3 Logical Expressions in WSML-Full ........................................................................ 13
   2.2 WSML-Core ...................................................................................................................... 16
      2.2.1 WSML-Core Syntax Basics .................................................................................... 17
      2.2.2 WSML-Core Ontologies ......................................................................................... 17
      2.2.3 WSML-Core Logical Expression Syntax ............................................................... 18
   2.3 WSML-DL ......................................................................................................................... 20
   2.4 WSML-Flight .................................................................................................................... 20
   2.5 WSML-Flight Syntax Basics ............................................................................................ 21
   2.6 WSML-Flight Ontologies ............................................................................................... 21
      2.6.1 WSML-Flight Logical Expression Syntax ............................................................ 21
   2.7 WSML-Rule ..................................................................................................................... 23
      2.7.1 WSML-Rule Logical Expression Syntax ............................................................... 24

3 **WSML-Semantics** ................................................................................................................. 26
   3.1 Mapping Conceptual Syntax to Logical Expression Syntax ........................................ 26
      3.1.1 Preprocessing steps................................................................................................. 29
      3.1.2 WSML-Core Semantics ......................................................................................... 30
      3.1.3 WSML-Flight Semantics ....................................................................................... 32
      3.1.4 WSML-Rule Semantics ......................................................................................... 34

**Conclusion** ............................................................................................................................... 37

**References** ............................................................................................................................. 39
INTRODUCTION

Ontologies and Semantic Web Services need formal languages for their specification in order to enable automatic processing. The W3C recommendation for an ontology language OWL [Dean & Schreiber, 2004] has serious limitations both on a conceptual level and with respect to some of its formal properties [de Bruijn et al., 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.

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

The advantage of having five WSML variants is that the user can decide for one language depending on the expressivity she needs and depending on the complexity of the algorithms dealing with the language. The languages are layered to a certain extent in order to offer the possibility to use a common subset or a common superset e.g. for mediation between ontologies formalized in different WSML variants. Especially for this purpose, WSML-Core and WSML-Full have been defined. WSML-Core is the least common subset of all WSML-variants having already the expressivity of plain Datalog. In contrast, WSML-Full which unifies all WSML variants under a common First-Order umbrella needs also non-monotonic extensions in order to capture the non-monotonic negation of WSML-Flight and WSML-Rule. Therefore, WSML-Full is clearly very expressive and the algorithms dealing with it will be very intricate. It is of no use to use WSML-Full in DIP because the language is far too expressive for real-world modelling tasks and the algorithms dealing with it will be far to intricate to be applicable for real-world tasks.

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.

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

1 The work presented in [Levy & Rousset, 1998] might serve as a starting point to define a subset of WSML-Full which could be used to enable a higher degree of interoperation between Description Logics and Logic Programming (while retaining decidability, but possibly losing tractability) than through their common core described in WSML-Core. If we would choose to minimize the interface between both paradigms, as described in [Eiter et al., 2004], it would be sufficient to add a simple syntactical construct to the Logic Programming language. This construct would stand for a query to the Description Logic knowledge base. Thus, the logic programming engine should have an interface to the Description Logic reasoner to issue queries and retrieve results.
The remainder of the document is structured as follows. In chapter 2 we will recapitulate the syntax of each of the WSML variants, i.e. also of WSML-Full. Then, in chapter 3, we will provide a semantic grounding of the syntax of each of the WSML variants.
2 SYNTAX OF LOGICAL EXPRESSIONS

In this chapter we will recapitulate the logical expressions syntax of the WSML variants. The syntax has been presented in the DIP deliverable D2.7. We start with the syntax of WSML-Full which will be restricted for the other WSML variants.

2.1 WSML-Full

In the following we will present ontology specifications and capability specifications within WSML-Full.

2.1.1 Ontology Specifications in WSML-Full

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:

```plaintext
ontology family
```

An ontology specification document in WSML consists of:

```
ontology = 'ontology' id? header* ontology_element*

ontology_element = concept | relation | instance | relationinstance | axiom
```

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

2.1.1.1 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.

<table>
<thead>
<tr>
<th>concept</th>
<th>=</th>
<th>'concept' id superconcept? nfp? attribute*</th>
</tr>
</thead>
<tbody>
<tr>
<td>superconcept</td>
<td>=</td>
<td>'subConceptOf' idlist</td>
</tr>
</tbody>
</table>

Example:

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

WSML allows creation of axioms in order to refine the definition already given in the conceptual syntax, i.e., the subconcept and attribute definitions. It is advised in the WSML specification to include the relation between the concept and the axioms related to the concept in the 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
declaration should be of the form:

\[ ?x \text{memberOf } A \text{ equivalent } ?x \text{memberOf } B_i \text{ and } \ldots \text{ and } ?x \text{memberOf } B_n \]

With \( A \) and \( B_1, \ldots, B_n \) concept identifiers.

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

\begin{verbatim}
axiom humanDefinition
  definedBy
    ?x memberOf Human equivalent ?x memberOf Primate and ?x memberOf LegalAgent.
\end{verbatim}

\textit{Attributes}

WSML allows two kinds of attribute definitions, namely, constraining definitions with
the keyword \texttt{ofType} and inferring definitions with the keyword \texttt{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 \texttt{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 \texttt{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 \texttt{ofType} and \texttt{impliesType} coincide, because datatypes have a known
domain and thus values cannot be inferred to be of a certain datatype.

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

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

<table>
<thead>
<tr>
<th>attribute</th>
<th>=</th>
<th>[attr]: id attributefeature* att_type cardinality? [type]: idlist nfp?</th>
</tr>
</thead>
<tbody>
<tr>
<td>att_type</td>
<td>=</td>
<td>'ofType'</td>
</tr>
<tr>
<td>cardinality</td>
<td>=</td>
<td>'(' min_cardinality]: digit+ [max_cardinality]: cardinality_number? ')'</td>
</tr>
<tr>
<td>cardinality_number</td>
<td>=</td>
<td>{finite_cardinality} digit+</td>
</tr>
<tr>
<td>attributefeature</td>
<td>=</td>
<td>'transitive'</td>
</tr>
</tbody>
</table>

When an attribute is specified as being transitive, this means that if three individuals $a$, $b$ and $c$ are related via a transitive attribute $att$ in such a way: $a$ $att$ $b$ $att$ $c$ then $c$ is also a value for the attribute $att$ at $a$: $a$ $att$ $c$.

When an attribute is specified as being symmetric, this means that if an individual $a$ has a symmetric attribute $att$ with value $b$, then $b$ also has attribute $att$ with value $a$.

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

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

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

2.1.1.2 Relations

A relation definition starts with the relation keyword, which is optionally followed by the identifier of the relation. WSML allows the specification of relations with arbitrary arity. The domain of the parameters can be optionally specified using the keyword impliesType or ofType. Note that parameters of a relation are strictly ordered. A relation definition is 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.

<table>
<thead>
<tr>
<th>relation</th>
<th>=</th>
<th>'relation' id arity? paramtyping? superrelation? nfp?</th>
</tr>
</thead>
<tbody>
<tr>
<td>arity</td>
<td>=</td>
<td>'/' pos_integer</td>
</tr>
<tr>
<td>paramtyping</td>
<td>=</td>
<td>'(' paramtype moreparamtype* ')'</td>
</tr>
<tr>
<td>paramtype</td>
<td>=</td>
<td>att_type idlist</td>
</tr>
<tr>
<td>moreparamtype</td>
<td>=</td>
<td>',', paramtype</td>
</tr>
<tr>
<td>superrelation</td>
<td>=</td>
<td>'subRelationOf' idlist</td>
</tr>
</tbody>
</table>

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 dc#relation.

2.1.1.3 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.

<table>
<thead>
<tr>
<th>instance</th>
<th>=</th>
<th>'instance' id? memberof? nfp? attributevalue*</th>
</tr>
</thead>
<tbody>
<tr>
<td>memberof</td>
<td>=</td>
<td>'memberOf' idlist</td>
</tr>
<tr>
<td>attributevalue</td>
<td>=</td>
<td>id 'hasValue' valuelist</td>
</tr>
</tbody>
</table>

Example:

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

Instances explicitly specified in an ontology are those which are shared together as part of the ontology. However, most instance data exists outside the ontology in private data stores. Access to these instances, as described in [Roman et al., 2004], is 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.
Below is an example of an instance of a ternary relation (note that the identifier is optional):

relationInstance = distance(Innsbruck, Munich, 234)

### 2.1.1.4 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 one or more 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 2.1.3.

Example of a defining axiom:

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

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

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

### 2.1.2 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.

### 2.1.2.1 Capabilities

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

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

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

Below is an example of a capability specified in WSML:

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

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

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

2.1.2.2 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.

\[
\begin{array}{|c|}
\hline
\text{interfaces} = | \text{interface} \\
| \text{minterfaces} \\
\text{minterfaces} = | \text{interface}'{"id moreids* '}' \\
\text{interface} = | \text{interface}'id? header* choreography? orchestration? \\
\text{choreography} = | \text{choreography}'id \\
\text{orchestration} = | \text{orchestration}'id \\
\hline
\end{array}
\]

Below is an example of an interface and an example of references to multiple interfaces:
We do not define ways to specify the choreography and orchestration here. Instead, we refer the reader to the corresponding WSMO deliverable D14 [Roman et al., 2005].

2.1.3 Logical Expressions in WSML-Full

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

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

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

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

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

**Definition 2.1.** A vocabulary $V$ of a WSML language $L(V)$ consists of the following:
• A set of identifiers $V_{ID}$.
• A set of object constructors $V_O \subseteq V_{ID}$.
• A set of function symbols $V_F \subseteq V_O$.
• A set of datatype wrappers $V_D \subseteq V_O$.
• A set of data values $V_{DV} \subseteq V_O$ which encompasses all string, integer and decimal values.
• A set of anonymous identifiers $V_A \subseteq V_O$ of the form _#, _#1, _#2, etc....
• A set of relation identifiers $V_R \subseteq V_{ID}$.
• A set of variable identifiers $V_V \subseteq V_{ID}$ of the form ?X.

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

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

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

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

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

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

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

- If $r \in V_R$ and $t_1, ..., t_n$ are terms, then $r(t_1, ..., t_n)$ is an atomic formula in $L(V)$.
- If $\alpha, \beta \in Term(V)$ then $\alpha = \beta$, $\alpha ::= \beta$ and $\alpha \neq \beta$ are atomic formulae in $L(V)$.
- If $\alpha, \beta \in Term(V)$ and $\gamma \in Term(V)$ or $\gamma$ is of the form \{ $\gamma_1, ..., \gamma_n$ \}, with $\gamma_1, ..., \gamma_n \in Term(V)$, then:
  - $\alpha$ subConceptOf $\gamma$ is an atomic formula in $L(V)$
  - $\alpha$ memberOf $\gamma$ is an atomic formula in $L(V)$
  - $\alpha[\beta$ ofType $\gamma]$ is an atomic formula in $L(V)$
  - $\alpha[\beta$ impliesType $\gamma]$ is an atomic formula in $L(V)$
  - $\alpha[\beta$ hasValue $\gamma]$ is an atomic formula in $L(V)$
Given the atomic formulae, we recursively define the set of formulae in $L(V)$ in definition 2.4.

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

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

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

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

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

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

> Human subConceptOf Mammal

> and Human[hasName ofType foaf#name] and Human[hasChild impliesType Human] Human

can be written as

> Human[hasName ofType foaf#name, hasChild impliesType Human] subConceptOf Mammal

The following are examples of WSML logical expressions (note that variables are implicitly universally quantified):
No human can be both male and female:

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

A human who is not a man is a woman:

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

The brother of a parent is an uncle:

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

Do not trust strangers:

\[?\text{x[distrust hasValue ?y]} \text{:- naf ?x[knows hasValue ?y]}.\]

A complex example:

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

2.2 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 [Grossof 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 [Grossof 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.
2.2.1 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 IRIs and are pairwise disjoint.
- The set of attribute names is equivalent to $V_R$
- The set of relation identifiers $V_R$ is split into two disjoint sets, $V_{RA}$ and $V_{RC}$, which correspond to relations with an abstract and relations with a concrete range, respectively.

2.2.2 WSML-Core Ontologies

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

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

2.2.2.1 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 inversty 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 inversty 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.

2.2.2.2 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.

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

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

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

2.2.2.3 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.

2.2.2.4 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.

2.2.3 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 $V_D$, $\varphi \in V_I$, $\psi$ 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 $L(V)$
- $\varphi \text{memberOf } \Gamma$ is an atomic formula in $L(V)$
- $\varphi[ p \text{ ofType } \Delta ]$ is an atomic formula in $L(V)$
- $\varphi[ s \text{ impliesType } \Delta ]$ is an atomic formula in $L(V)$
- $\varphi[ s \text{ impliesType } \Gamma ]$ is an atomic formula in $L(V)$
- $\varphi[ s \text{ hasValue } Val ]$ is an atomic formula in $L(V)$

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

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

In the following, $F$ stands for an lhs-formula, 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

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

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

- Any a-molecule is an rhs-formula
- if $G$ and $H$ are lhs-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 $L(V)$.
- $Var_i[ p \text{ hasValue } Var_2 ] \text{ impliedBy } Var_i[ p \text{ hasValue } Var_3 ] \text{ and } Var_3[ p \text{ hasValue } Var_2 ]$ (globally transitive attribute/relation) is a formula in $L(V)$.  

19
• \( \text{Var}_1[p \ \text{hasValue} \ \text{Var}_2] \ \text{impliedBy} \ \text{Var}_2[p \ \text{hasValue} \ \text{Var}_1] \) (globally symmetric attribute/relation) is a formula in \( L(V) \).

• \( \text{Var}_1[p \ \text{hasValue} \ \text{Var}_2] \ \text{impliedBy} \ \text{Var}_1[q \ \text{hasValue} \ \text{Var}_2] \) (globally sub-attribute/relation)) is a formula in \( L(V) \).

• \( \text{Var}_1[p \ \text{hasValue} \ \text{Var}_2] \ \text{impliedBy} \ \text{Var}_2[q \ \text{hasValue} \ \text{Var}_1] \) (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.

• \( H \ \text{impliedBy} \ F \) is a formula in \( L(V) \) if all the WSML variables occurring in \( H \) occur in \( F \) as well and the variable graph of \( F \) is connected and acyclic.

• If \( \text{Head} \ \text{impliedBy} \ \text{Body} \) is a formula in \( L(V) \), then \( \text{Body} \ \text{implies} \ \text{Head} \) a formula in \( L(V) \).

• Any occurrence of a molecule of the form \( \text{Var}_1[p \ \text{hasValue} \ \text{Var}_2] \) in a WSML-Core clause can be interchanged with \( p(\text{Var}_1, \text{Var}_2) \) (i.e., these two forms can be used interchangably in WSML Core).

Here, the variable graph of a logical expression \( E \) is defined as the undirected graph having all WSML variables in \( E \) as nodes and an edge between \( \text{Var}_2 \) and \( \text{Var}_2 \) for every molecule \( \text{Var}_1[p \ \text{hasValue} \ \text{Var}_2] \), or \( p(\text{Var}_1, \text{Var}_2) \), respectively.

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.

2.3 WSML-DL

To be included

2.4 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.

2.5 WSML-Flight Syntax Basics
WSML-Flight adheres to the WSML syntax basics. The restrictions posed on these syntax basics by WSML-Core do not apply to WSML-Flight.

2.6 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 2.1.1.

Note that for axioms, we only allow a restricted form of logical expressions, as defined in Section 2.6.1.

2.6.1 WSML-Flight logical Expression Syntax

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

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

The syntax for logical expressions of WSML Flight is the same as described in Section 2.1.3 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 TermFlight(V) for a given vocabulary V.

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

- Any f ∈ VO is a term.
- Any v ∈ VV is a term
- If d ∈ VD and dv₁, ..., dvₙ are in VDV ∪ VV, then d(dv₁, ..., dvₙ) 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 $\alpha \neq \beta$ 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:

- Given a head-formula $\beta \in \text{Head}(V)$ and a body-formula $\alpha \in \text{Body}(V)$, $\beta :- \alpha$ 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, (3) the safety condition holds and (4) the resulting WSML-Flight knowledge base is locally stratified.
- We define the set of admissible head formulae $\text{Head}(V)$ as follows:
  - Any atomic formula $\alpha$ which does not contain the inequality symbol ($\neq$) or the unification operator ($=$) is in $\text{Head}(V)$.
  - Let $\alpha, \beta \in \text{Head}(V)$, then $\alpha$ and $\beta$ is in $\text{Head}(V)$.
  - $\alpha \text{ implies } \beta$ is in $\text{Head}(V)$.
  - $\alpha \text{ impliedBy } \beta$ is in $\text{Head}(V)$.
  - $\alpha \text{ equivalent } \beta$ is in $\text{Head}(V)$.
- Any 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 $\alpha \in \text{Body}(V)$, $\text{naf } \alpha$ 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 $\text{Body}(V)$.
- Any formula of the form $\neg \alpha$ with $\alpha \in \text{Body}(V)$ is an admissible formula and is called a constraint.

As with the general WSML logical expression syntax, $\text{<-}$, $\text{->}$ and $\text{<->}$ can be seen as synonyms of the keywords $\text{implies}$, $\text{impliedBy}$ and $\text{equivalent}$, 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 $A_2$ :- $B$
  - $A_1$ impliesBy $A_2$ :- $B$

- Rules of the form $A_1$ impliedBy $A_2$ :- $B$ are transformed to:
  - $A_1$ :- $A_2$ and $B$

- Rules of the form $A_1$ implies $A_2$ :- $B$ are transformed to:
  - $A_2$ :- $A_1$ and $B$

- Rules of the form $A$ :- $B_1$ or ... or $B_n$ are split into $n$ different rules:
  - $A$ :- $B_1$
  - ...
  - $A$ :- $B_n$

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

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

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

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

The following are examples of WSML-Flight logical expressions:

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

2.7 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.
2.7.1 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 2.1.3 with the restrictions which are described in the following: we define the notion of a WSML-Rule vocabulary in Definition 6.1.

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

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

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

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

Definition 6.3. Given a set of WSML-Rule terms TermRule(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 \( \alpha, \beta \in \text{TermRule}(V) \) then \( \alpha = \beta \) and \( \alpha \neq \beta \) are atomic formulae in L(V).
- If \( \alpha, \beta \in \text{TermRule}(V) \) and \( \gamma \in \text{Term}(V) \) or \( \gamma \) is of the form \( \{ \gamma_1, \ldots, \gamma_n \} \) with \( \gamma_1, \ldots, \gamma_n \in \text{TermRule}(V) \), 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:

- Given a head-formula \( \beta \in \text{Head}(V) \) and a body-formula \( \alpha \in \text{Body}(V) \), \( \beta :\alpha \) 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, (3) the safety condition holds and (4) the resulting WSML-Rule knowledge base is locally stratified.
- We define the set of admissible head formulae Head(V) as follows:
  - Any atomic formula \( \alpha \) which does not contain the inequality symbol (\( \neq \)) or the unification operator (\( = \)) is in Head(V).
Let $\alpha, \beta \in \text{Head}(V)$, then $\alpha$ and $\beta$ is in $\text{Head}(V)$.

- $\alpha$ implies $\beta$ is in $\text{Head}(V)$.
- $\alpha$ impliedBy $\beta$ is in $\text{Head}(V)$.
- $\alpha$ equivalent $\beta$ is in $\text{Head}(V)$.

- Any 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 $\alpha \in \text{Body}(V)$, $\text{naf} \ \alpha$ 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 $\text{Body}(V)$.

- Any formula of the form $\neg \alpha$ with $\alpha \in \text{Body}(V)$ is an admissible formula and is called a constraint.

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

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

The following are examples of WSML-Rule logical expressions:

- $f(?y) \text{ memberOf } ?z \text{ impliedBy } ?z \text{ memberOf } ?x :- \text{naf } ?y[a \text{ hasValue } f(g(?x))], \text{start hasValue } _\text{date}(2005,6,6,0,0), \text{nr hasValue } 10, \text{name hasValue "myName"}]$ and $p(?z)$. 

3 WSML-SEMANTICS

3.1 Mapping Conceptual Syntax to Logical Expression Syntax

In the following we present the semantics of WSML which has syntactically been specified in the DIP deliverable D2.7. There, we have defined the conceptual and logical expression syntax for different WSML variants. In the following we specify the formal semantics for the WSML variants, in this version of the document of WSML-Core, WSML-Flight and WSML-Rule. Specification of the semantics of WSML-DL constitutes future work. As we disadvise the usage of WSML-Full for DIP, we don’t provide the WSML-Full semantics in this deliverable. Note that the semantics of capability descriptions is not entirely clear at the moment. Therefore, we only define the semantics of ontology definitions.

In the following we provide first the syntax specifications of the WSML variants that are relevant for DIP, namely WSML-Core, WSML-Flight, WSML-Rule and WSML-DL. We don’t provide the syntax specification of WSML-Full because we have disadvised the usage of this

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

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 $\text{subRelationOf}$. E.g., "$P \text{subRelationOf} \{Q, R\}" is substituted by "$P \text{subRelationOf} Q" and "$P \text{subRelationOf} R$".
- Expand all sQNames to full IRIs using the namespace declarations.

Table 8.1 contains the mapping between the WSML conceptual syntax for ontologies and the logical expression syntax through the mapping function $\tau$ ($X$ and $Y$ are metavariables 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 of the DIP deliverable D2.7) and boldfaced keywords refer to keywords in the WSML language.

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

\( \tau(\text{ontology}) \)  
\( \tau(\text{concept id superconcept attribute}_1 ... attribute_n) \)  
\( \tau(\text{subConceptOf idlist, X}) \)  
\( \tau(\text{attribute id impliesType feature cardinality range idlist, X}) \)  
\( \tau(\text{transitive, X, Y}) \)  
\( \tau(\text{symmetric, X, Y}) \)  
\( \tau(\text{reflexive, X, Y}) \)  
\( \tau(\text{inverseOf att_id, X, Y}) \)  
\( \tau(\text{(n), X, Y}) \)  
\( \tau(\text{(n m), X, Y}) \)  
\( \tau(\text{(n *), X, Y}) \)  
\( \tau(\text{(0 m), X, Y}) \)  
\( \tau(\text{relation id/arity superrelation}) \)  
\( \tau(\text{subRelationOf id, X, Y}) \)  
\( \tau(\text{instance id memberof attributevalue}_1 ... attributevalue_n) \)

WSML Logical Expression Syntax

\( \tau(\text{ontology}_{element_1}) ... \tau(\text{ontology}_{element_n}) \)  
\( \tau(\text{superconcept, id}) \tau(\text{attribute}_1, id) ... \tau(\text{attribute}_n, id) \)  
\( X \text{ subConceptOf idlist.} \)  
\( \tau(\text{cardinality}, X, \text{attribute id}) \)  
\( \text{?x memberOf } X \text{ and ?x[attribute id hasValue y] implies ?y memberOf range idlist.} \)  
\( \text{?y memberOf range idlist.} \)  
\( \text{?x memberOf X implies ?x[Y hasValue y] and naf ?y memberOf range idlist.} \)  
\( \text{?x memberOf X and ?x[Y hasValue y] implies ?y[Y hasValue z].} \)  
\( \text{?x memberOf X implies ?y[Y hasValue x].} \)  
\( \text{?x memberOf X and ?x[Y hasValue y] implies ?y[att_id hasValue x].} \)  
\( \text{?y memberOf X and ?x[att_id hasValue y] implies ?y[Y hasValue x].} \)  
\( \tau(\text{(n), X, Y}) \)  
\( \tau(\text{(n m), X, Y}) \)  
\( \tau(\text{(n *), X, Y}) \)  
\( \tau(\text{(0 m), X, Y}) \)  
\( \tau(\text{superrelation, id, arity}) \)  
\( X(?x_1,...,?x_l) \text{ implies id(?x_1,...,?x_l).} \)  
\( \tau(\text{memberof id, attributevalue}_1 ... attributevalue_n) \)  
\( \tau(\text{memberof id, attributevalue}_1 ... attributevalue_n) \)
Table 8.1: Mapping WSML conceptual syntax to logical expression syntax.

<table>
<thead>
<tr>
<th>WSML Conceptual Syntax</th>
<th>WSML Logical Expression Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$(memberOf $idlist$, $X$)</td>
<td>$X$ memberOf $idlist$.</td>
</tr>
<tr>
<td>$\tau$(att_id hasValue valuelist, $X$)</td>
<td>$X$[att_id hasValue valuelist].</td>
</tr>
<tr>
<td>$\tau$(axiom id log_expr)</td>
<td>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"}

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

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

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

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

To the following logical expressions:

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

!- naf ?x memberOf _"http://www.example.org/ontologies/example#Human" and
   _"http://www.example.org/ontologies/example#ageOfHuman"(?x,?y).
!- naf ?y memberOf _integer and
   _"http://www.example.org/ontologies/example#ageOfHuman"(?x,?y).
```
3.1.1 Preprocessing steps

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

Replacing idlists with multiple statements

Statements involving argument lists of the form $A \ op \ {v_1, \ldots, v_n}$, with $op \in \{\text{hasValue}, \ ofType, \ impliesType\}$, 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 $is-a \in \{\text{memberOf}, \ subConceptOf\}$, 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 $is-a \in \{\text{memberOf}, \ subConceptOf\}$ and $op_i \in \{\text{hasValue}, \ ofType, \ impliesType\}$ 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 $op_i \in \{\text{hasValue}, \ ofType, \ impliesType\}$, are translated to: $a[c_1 \ op_1 \ d_1] \ and \ ... \ and \ a[c_n \ op_n \ d_n]$

Replacing equivalence with two implications

$$lexpr \ equivalent \ rexpr. := lexpr \ implies \ rexpr. \ lexpr \ impliedBy \ rexpr.$$

Replacing right implication with left implication.

$$lexpr \ implies \ rexpr. := rexpr \ impliedBy \ lexpr.$$

Rewriting data term shortcuts

The shortcuts for writing strings, integers and decimals are rewritten to their full form:

- "$string" := _string("string") \ (unless "$string" already occurs in the _string datatype wrapper)
- $integer := _integer("integer")$
- $decimal := _decimal("decimal")$

Rewrite data terms to predicates

Data terms occur as functions in WSML. However, Datalog does not allow the use of function symbols. Thus, we rewrite the datatype wrappers to built-in predicates as follows:

Each datatype wrapper with arity $n$ has a corresponding built-in predicate with
the same name as the datatype wrapper (cf. Appendix C of the DIP deliverable D2.7). This built-in predicate always has an arity \( n+1 \). Each occurrence of a datatype wrapper \( \delta \) in a statement \( \varphi \) is replaced with a new variable \(?x\) and the datatype predicate corresponding to the wrapper \( \delta \) is conjoined with the resulting statement \( \varphi' \): (\( \varphi' \) and \( \delta(X_1,\ldots,X_i,x) \)).

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 of the DIP deliverable D2.7 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.

### 3.1.2 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(\text{head impliedBy body.}) )</td>
<td>( \pi(\text{head}) \leftarrow \pi(\text{body}) )</td>
</tr>
<tr>
<td>( \pi(\text{lexpr or rexpr}) )</td>
<td>( \pi(\text{lexpr}) \lor \pi(\text{rexpr}) )</td>
</tr>
<tr>
<td>( \pi(\text{lexpr and rexpr}) )</td>
<td>( \pi(\text{lexpr}) \land \pi(\text{rexpr}) )</td>
</tr>
</tbody>
</table>
Table 8.2: WSML-Core Semantics

<table>
<thead>
<tr>
<th>WSML</th>
<th>Horn logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\pi(Xl , \text{memberOf} , id2))</td>
<td>(id2(Xl))</td>
</tr>
<tr>
<td>(\pi(id1 , \text{subConceptOf} , id2))</td>
<td>(id2(x) \leftarrow id1(x))</td>
</tr>
<tr>
<td>(\pi(Xl[id2 , \text{hasValue} , X2]))</td>
<td>(id2(Xl,X2))</td>
</tr>
<tr>
<td>(\pi(id1[id2 , \text{impliesType} , id3]))</td>
<td>(id3(y) \leftarrow id1(x) \land id2(x,y))</td>
</tr>
<tr>
<td>(\pi(id1[id2 , \text{ofType} , dt]))</td>
<td>(dt(y) \leftarrow id1(x) \land id2(x,y))</td>
</tr>
<tr>
<td>(\pi(p(X_l,...,X_n)))</td>
<td>(p(X_l,...,X_n))</td>
</tr>
</tbody>
</table>

Note that in the translation of typing constraints using \textbf{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: \(\leftarrow id1(x) \land id2(x,y) \land \text{not} \, dt(y)\), with 'not' being default negation. It is the same for the use of \textbf{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 \(A_1 \land ... \land A_n \leftarrow B\) are split in \(n\) different rules:
  - \(A_1 \leftarrow B\)
  - ...
  - \(A_n \leftarrow B\)
- Rules of the form \(A_1 \leftarrow A_2 \leftarrow B\) are transformed to:
  - \(A_1 \leftarrow A_2 \land B\)
- Rules of the form \(A \leftarrow B_1 \lor ... \lor B_n\) are split into \(n\) different rules:
  - \(A \leftarrow B_1\)
  - ...
  - \(A \leftarrow B_n\)

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

**Definition 8.3 (Entailment in WSML-Core)** We say a WSML-Core knowledge base \(KB_A\) entails a WSML-Core knowledge base \(KB_B\), written as: \(KB_A \models_{\text{WSML}} KB_B\) iff \(\pi(KB_A) \models \pi(KB_B)\), where \(\models\) is the classical entailment relation.
3.1.3 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 the Datalog fragment of F-Logic [Kifer et al., 1995] (extended with inequality and stratified default negation in the body of the rule) using the mapping function $\pi$.

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 default negation and a rule with an empty head denoting an integrity constraint):

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

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

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

\[ v:z \leftarrow _\text{impliestype}(x,y,z) \land w:x \land w[y \rightarrow v] \]

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

<table>
<thead>
<tr>
<th>WSML</th>
<th>F-Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi(!- \text{ body.})$</td>
<td>$\leftarrow \pi(\text{body})$</td>
</tr>
<tr>
<td>$\pi(\text{head ::= body.})$</td>
<td>$\pi(\text{head}) \leftarrow \pi(\text{body})$</td>
</tr>
<tr>
<td>$\pi(\text{lexpr impliedBy rexpr.})$</td>
<td>$\pi(\text{lexpr}) \leftarrow \pi(\text{rexpr})$</td>
</tr>
<tr>
<td>$\pi(\text{lexpr or rexpr})$</td>
<td>$\pi(\text{lexpr}) \lor \pi(\text{rexpr})$</td>
</tr>
<tr>
<td>$\pi(\text{lexpr and rexpr})$</td>
<td>$\pi(\text{lexpr}) \land \pi(\text{rexpr})$</td>
</tr>
<tr>
<td>$\pi(X1 \text{ memberOf X2})$</td>
<td>$X1:X2$</td>
</tr>
<tr>
<td>$\pi(X1 \text{ subConceptOf X2})$</td>
<td>$X1::X2$</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>$\pi(X_1[X_2 \text{ hasValue } X_3])$</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>$\pi(X_i = X_2)$</td>
<td>$X_i = X_2$</td>
</tr>
<tr>
<td>$\pi(X_i \neq X_2))$</td>
<td>$X_i \neq X_2$</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. All integrity constraints are part of the set of integrity constraints $C$.

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

Application of the usual Lloyd-Topor transformations [Lloyd and Topor, 1984] 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 into $n$ different rules:
  - $A_1 \leftarrow B$
  - ...
  - $A_n \leftarrow B$
- Rules of the form $A_1 \leftarrow A_2 \leftarrow B$ are transformed to:
  - $A_1 \leftarrow A_2 \land B$
- Rules of the form $A \leftarrow B_{i,\ldots,B_n}$ are split into $n$ different rules:
  - $A \leftarrow B_i$
  - ...
  - $A \leftarrow B_n$

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

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

### 3.1.4 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 (locally) stratified negation. However, the mapping looks exactly the same as Table 8.3. The only difference is that the meta-variables $X#$ can also stand for a constructed term.

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

**Definition 8.7.** We define a WSML-Rule knowledge base $KB$ as a collection of 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 3.2 to a WSML-Rule ontology.

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

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 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:

$$\leftarrow x[y \Rightarrow z] \land w:x \land w[y \rightarrow v] \land notv:z$$

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

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

Now follows the semantics of WSML-Rule in Table 8.4. In the table, $X#$ stands for either a variable, an identifier, or a constructed term; '=' is the unification operator and '!=' is the built-in inequality symbol.
Table 8.4: Semantics of WSML-Rule

<table>
<thead>
<tr>
<th>WSML F-Logic</th>
<th>F-Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\pi(!\text{- body}))</td>
<td>(\pi(\text{body}))</td>
</tr>
<tr>
<td>(\pi(\text{head} \leftarrow \text{body}))</td>
<td>(\pi(\text{head}) \leftarrow \pi(\text{body}))</td>
</tr>
<tr>
<td>(\pi(\text{expr impliedBy reexpr}))</td>
<td>(\pi(\text{expr}) \leftarrow \pi(\text{reexpr}))</td>
</tr>
<tr>
<td>(\pi(\text{expr or reexpr}))</td>
<td>(\pi(\text{expr}) \lor \pi(\text{reexpr}))</td>
</tr>
<tr>
<td>(\pi(\text{expr and reexpr}))</td>
<td>(\pi(\text{expr}) \land \pi(\text{reexpr}))</td>
</tr>
<tr>
<td>(\pi(\text{X1 memberOf X2}))</td>
<td>(\text{X1} : \text{X2})</td>
</tr>
<tr>
<td>(\pi(\text{X1 subConceptOf X2}))</td>
<td>(\text{X1} :: \text{X2})</td>
</tr>
<tr>
<td>(\pi(\text{X1[X2 hasValue X3]}))</td>
<td>(\text{X1[X2} \rightarrow \text{X3]})</td>
</tr>
<tr>
<td>(\pi(\text{X1[X2 ofType X3]}))</td>
<td>(\text{X1[X2} \Rightarrow \text{X3]})</td>
</tr>
<tr>
<td>(\pi(\text{X1[X2 impliesType X3]}))</td>
<td>_(impliestype(X1,X2,X3))</td>
</tr>
<tr>
<td>(\pi(p(X_1,...,X_n)))</td>
<td>(p(X_1,...,X_n))</td>
</tr>
<tr>
<td>(\pi(\text{X1} = \text{X2}))</td>
<td>(\text{X1} = \text{X2})</td>
</tr>
<tr>
<td>(\pi(\text{X1} != \text{X2}))</td>
<td>(\text{X1} != \text{X2})</td>
</tr>
</tbody>
</table>

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 \(\text{ofType}\). All integrity constraints are part of the set of integrity constraints \(C\).

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

Application of the usual Lloyd-Topor transformations [Lloyd and Topor, 1984] 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 \(A_1 \land ... \land A_n \leftarrow B\) are split into \(n\) different rules:
  - \(A_1 \leftarrow B\)
  - ...
  - \(A_n \leftarrow B\)
- Rules of the form \(A_1 \leftarrow \text{LT} A_2 \leftarrow B\) are transformed to:
  - \(A_1 \leftarrow A_2 \land B\)
- Rules of the form \(A \leftarrow B_1 \lor ... \lor B_n\) are split into \(n\) different rules:
We base the semantics of WSML-Rule on the perfect model semantics by Przymusinski [Przymusinski, 1989], which defines the semantics for (locally) stratified logic programs. Przymusinski shows that every locally stratified program has a unique perfect model. WSML-Rule only allows locally stratified negation.

**Definition 8.8 (Satisfiability in WSML-Rule)** Let $KB$ be a WSML-Rule knowledge base and $C$ be a set of constraints. $KB$ is satisfiable iff $\pi(KB)$ has a perfect model $M_{KB}$ which does not violate any of the constraints in $C$. We say an integrity constraint is violated if some ground instantiation of the 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 perfect model of $KB$.
CONCLUSION
In this deliverable we have presented the semantics of WSML-Core, WSML-Flight and WSML-Rule.
REFERENCES


