



D2v1.0. Web Service Modeling Ontology (WSMO)

WSMO Working Draft 16 August 2004

This version:

<http://www.wsmo.org/2004/d2/v1.0/20040816/>

Latest version:

<http://www.wsmo.org/2004/d2/v1.0/>

Previous version:

<http://www.wsmo.org/2004/d2/v1.0/20040809/>

Editors:

Dumitru Roman
Holger Lausen
Uwe Keller

Authors:

Dumitru Roman
Holger Lausen
Uwe Keller
Eyal Oren
Christoph Bussler
Michael Kifer
Dieter Fensel

This document is also available in non-normative [PDF](#) version. The intent of the document is to be submitted to [W3C](#).

Abstract

This document presents an ontology called Web Service Modeling Ontology (WSMO) for describing various aspects related to Semantic Web Service. Having the Web Service Modeling Framework (WSMF) as a starting point, we refine this framework and develop a formal ontology and language.

Table of contents

- [1. Introduction](#)
- [2. Global Issues](#)
 - [2.1 Namespaces](#)
 - [2.2 Identifier](#)
 - [2.3 Datatypes](#)
 - [2.4 Informal language for describing the WSMO elements](#)
 - [2.5 Non functional properties](#)
 - [2.5.1 Non functional properties - core properties](#)

- [2.5.2 Non functional properties - web service specific properties](#)
 - [3. Ontologies](#)
 - [3.1 Non functional properties](#)
 - [3.2 Import Ontologies](#)
 - [3.3 Used mediators](#)
 - [3.4 Concepts](#)
 - [3.5 Relations](#)
 - [3.6 Functions](#)
 - [3.7 Instances](#)
 - [3.8 Axioms](#)
 - [4. Goals](#)
 - [5. Mediators](#)
 - [6. Web Services](#)
 - [6.1 Capability](#)
 - [6.2 Interfaces](#)
 - [7. Logical language for defining formal statements in WSMO](#)
 - [8. Conclusions and further directions](#)
 - [References](#)
 - [Acknowledgement](#)
 - [Appendix A. Conceptual Elements of WSMO](#)
 - [Appendix B. BNF Grammar for WSML](#)
-

1. Introduction

WSMF [[Fensel & Bussler, 2002](#)] consists of four different main elements for describing semantic web services (see [Figure 1](#)): ontologies that provide the terminology used by other elements, goals that define the problems that should be solved by web services, web services descriptions that define various aspects of a web service, and mediators which bypass interpretability problems.

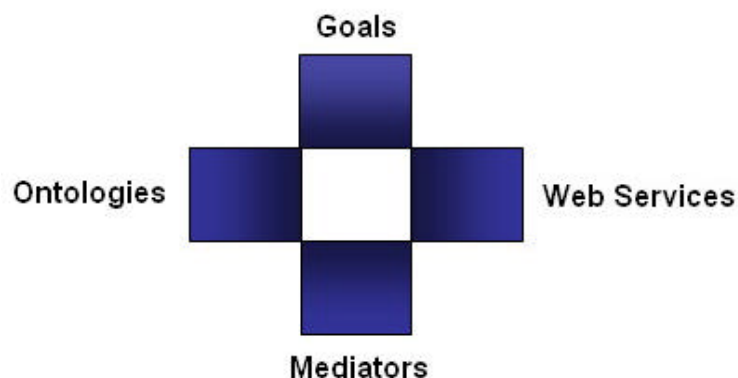


Figure 1. The main elements of WSMF

This document presents an ontology called Web Service Modeling Ontology (WSMO) for describing various aspects related to Semantic Web Service. Having the Web Service Modeling Framework (WSMF) as a starting point, we refine this framework and develop a formal ontology and language.

[Section 2](#) presents global issues related to various elements that are considered in WSMO. Following the philosophy of WSMF, we further define in the next sections the ontologies ([Section 3](#)), goals ([Section 4](#)), mediators ([Section 5](#)) and web service ([Section 6](#)). In [Section 7](#) we define the syntax of the logical language that is used in WSMO. The semantics and computationally tractable subsets of this logical language are defined and

discussed by the [WSML working group](#). [Section 8](#) presents our conclusions and further directions.

For a brief tutorial on WSMO we refer to the [WSMO Primer](#) [Arroyo & Stollberg, 2004] and for a non-trivial use case demonstrating how to use WSMO in a real-world setting we refer to the [WSMO Use Case Modeling and Testing](#) [Stollberg et al., 2004].

Besides the [WSMO working group](#) there are two more working groups related to the WSMO initiative: The [WSML working group](#) focusing on language issues and developing a adequate Web Service Modeling Language with various sublanguages as well the [WSMX working group](#) that is concerned with designing and building a reference implementation of an execution environment for WSMO.

2. Global Issues

This section addresses issues that concern all subsequent sections, it is especially concerned with compliance to current web standards.

2.1 Namespaces

The vocabulary is fully extensible, being based on URIs with optional fragment identifiers (URI references, or URIsrefs) [Berners-Lee et al, 1998]. URI references are used for naming all kinds of things in WSMO. Subsequent we allow the use of namespace abbreviations and its declaration.

The default namespace for the listings in this document is wsmo: <http://www.wsmo.org/2004/d2/>. Furthermore this document assumes the following namespace declarations:

- dc: <http://purl.org/dc/elements/1.1#>
- xsd: <http://www.w3.org/2001/XMLSchema#>
- foaf: <http://xmlns.com/foaf/0.1/>

In the remainder of this document the above prefixes denote the correspondingly defined URIs.

2.2 Identifier

WSMO distinguishes 4 kinds of identifiers: URI references, literals, anonymous Ids and variable names.

URI references

Everything in WSMO is an Identifier denoted by a URI, except when it is a Literal, a Variable or an Anonymous Id. WSMO is based on the idea of identifying things using web identifiers (called Uniform Resource Identifiers). However that does not limit WSMO to make statements about things that are not accessible on the web, like with the uri: "urn:isbn:0-520-02356-0" that identifies a certain book. Also the WSMO keywords (defined in Appendix B) are URIs; for brevity the namespace prefix of WSMO keywords may be omitted. URIs can be expressed as follows:

- **full URIs:** e.g. <http://www.wsmo.org/2004/d2>
- **qualified Names (QNames)** that are resolved using namespace declarations. For more details on QNames, we refer to [Bray et al., 1999].

In order to explicitly distinguish full URIs syntactically from QNames, we always use angle brackets to envelope full URIs, e.g. `<http://www.wsmo.org/2004/d2>`.

Literals

Literals are used to identify values such as numbers by means of a lexical representation. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals. Literals are either plain literals or typed literals. A Literal can be typed to a data type (e.g. to `xsd:integer`). Formally such a data type is defined by [Hayes, 2004]:

- a non-empty set of character strings called the lexical space of `d`;
e.g. {"true", "1", "false", "0"};
- a non-empty set called the value space of `d`;
e.g. {true, false};
- a mapping from the lexical space of `d` to the value space of `d`, called the lexical-to-value mapping of `d`;
e.g. {"true", "1"}->{true}; {"false", "0"}->{false}.

Furthermore the data type may introduce facets on its value space, such as ordering and therefore define the axiomatization for the relations `<`, `>` and function symbols like `+` or `-`). These special relations and functions are discussed in [Section 2.3](#) as so called build-ins for datatypes.

Syntactically, in WSMO literals always start and ends with double quotes, e.g. "WSMO working group" or "http://www.wsmo.org/2004/d2". This allows to syntactically distinguish between URIs and literals. Typed literals are literals that follow above convention followed by `^^` and the datatype URI, e.g. "1^^xsd:integer".

For the sake of convenience, we allow additionally syntactic shortcuts such as `42` for `"42"^^xsd:integer`, `4.2` for `"4.2"^^xsd:float`. I.e., a number `x` without decimal point is by default assumed to stand short for `"x"^^xsd:integer` and a number with decimal point `x.y` is by default assumed to stand short for `"x.y"^^xsd:float`.

Anonymous Ids

Anonymous Ids can be numbered (`_#1`, `_#2`, ...) or unnumbered (`_#`). Anonymous Ids represent Identifier. The same numbered Anonymous Id represents the same Identifier within the same scope (`logicalExpression`), otherwise Anonymous Ids represent different Identifier [Yang & Kifer, 2003]. Anonymous Ids can be used to denote objects that exists, but don't need a specific identifier (e.g. if someone wants to say that a Person John has an address `_#` which itself has a street name "hitchhikerstreet" and a street number "42", then the object of the address itself does not need a particular URI, but since it must exist as connecting object between John and "hitchhikersstreet", "42" we can denote it with an Anonymous Id).

The concept of anonymous IDs is similar to blank nodes in RDF [Hayes, 2004], however there are some differences. Blank Nodes are essentially existential quantified variables, where the quantifier has the scope of one document. RDF defines different strategies for the union of two documents (merge and union), whereas the scope of one anonymous ID is a logical expression and the semantics of anonymous ids do not require different strategies for a union of two documents respectively two logical expressions. Furthermore Anonymous IDs are not existentially quantified variables, but distinct constants. This allows two flavors of entailment: Strict and Relaxed, where the relaxed entailment is equivalent to the behavior of blank nodes and the strict entailment allows an easier treatment wrt. implementation.

Variable names

Variable names are strings that start with a question mark '?', followed by any positive number of symbols in {`a-z`, `A-Z`, `0-9`, `_`, `-`}, i.e. `?var` or `?lastValue_Of`.

2.3 Datatypes

In this section we discuss the role of datatypes and build-in operators that accommodate them. As WSMO is aligning with current web standards, we define the initial set of datatypes according to the primitive XML Schema data types, allowing some syntactical shortcuts as described in Table 1:

For brevity we allow an abbreviated notation for typed literals, according to the following table:

Table 1. Short Notations for for data types

XSD data type	Pattern	Example
xsd:string	Any sequence of characters enclosed by single quotes.	'some string' instead of "some string" ^{^^xsd:string}
xsd:boolean	The character sequence "true" or "false".	true instead of "true" ^{^^xsd:boolean}
xsd:integer	Any sequence of digits without decimal point.	42 instead of "42" ^{^^xsd:integer}
xsd:float	Any sequence of digits separated by a decimal point.	4.2 instead of "4.2" ^{^^xsd:float}
xsd:date	A sequence of characters following exactly the "yyyy-mm-dd" pattern, where y,m,d are placeholders for any digit.	2004-02-33 instead of "2004-02-33" ^{^^xsd:date}

Note: The pattern defined in Table 1 are ambiguous with certain QNames that do not have a namespace prefix, in such a case the short cut notations have precedence before the QName resolution to an URI. To denote such a ambiguous URI a namespace prefix or a full URI has to be used, e.g. <http://example.org/2004-03-07> instead of 2004-03-07 (given <http://example.org/> is the default namespace).

Note: In opposite to [Hayes, 2004] we do allow `xsd:duration` and recommend to use it as defined in Annex [D.6](#) of [Malhotra et al., 2004].

In addition to data type URIs that denote XML Schema primitive datatypes, we do allow XML Schema simple types and XML Schema derived datatypes. The interpretation of all datatypes is external to the language and done by a data type oracle.

Built-in relations

Built in relations are denoted by relation symbols in the logical language and evaluated by an external data type oracle. A list of built-in relations is defined in [Malhotra et al., 2004] where they are referred as comparison functions. For WSML built-in relation symbols we reuse the [mapping](#) between the XQuery functions symbols to the XPath 2.0 operators. For the mapping between the comparison functions and relations holds the following: X1, X2 is a tuple of a wsml relation if and only if the range value of the corresponding comparison function for X1, X2 is true. In addition to the XPath 2.0 operator symbols we define the following synonyms for better readability:

Table 2. WSMML Build-In relations and their mapping to XPath Notation

XPath Comparison Function Symbol	WSML Relation Symbol
eq	=
ne	!=
lt	<
gt	>
le	<=
ge	>=

Built-in functions

Built-in functions are denoted by functions symbols in the logical language and evaluated by an external data type oracle. A list of those functions is defined in [Malhotra et al., 2004]. As for relation we reuse the operator [mapping](#) to XPath 2.0.

2.4 Informal language for describing the WSMO elements

This section describes in an informal way the meaning of the keywords marked with bold in the listings contained in this document; they are used in an intuitive way for describing WSMO elements and their properties:

- **entity** - used when a WSMO element is defined (e.g. **entity** X: X is defined as a WSMO element).
- **subEntityOf** - used when a WSMO element is defined to specify that it also has the properties of another WSMO element (e.g. **entity** X **subEntityOf** Y: WSMO element X has all the properties of WSMO element Y); this keyword is optional; all things followed by the declaration of an element (using **entity** and optionally **subEntityOf**) represent properties of this element.
- **ofType** - used to specify the possible types of the single value a property may have; the set of types is marked by "{" and "}" and the types are separated by "," (e.g. X **ofType**{Y, Z}: X has a single value which can be either of type Y or Z); **ofType** keyword is optional.
- **ofTypeSet** - used to specify the possible types of the multiple values a property may have; the set of types is marked by "{" and "}" and the types are separated by "," (e.g. X **ofTypeSet** {Y, Z}: X might have multiple values which can be of type Y or Z); **ofTypeSet** this keyword is optional.

2.5 Non functional properties

We classify non functional properties in two categories: core properties and web service specific properties.

2.5.1 Non functional properties - core properties

The core properties can be used for all the modeling elements of WSMO. They consist of the [Dublin Core Metadata Element Set](#) [Weibel et al., 1998] plus the `version` element. We do not enforce restrictions on the range value type (and this also omit the range restriction in the following listings) but in some cases provide additional recommendations. In case that no WSMO recommendation is given, the Dublin Core rules apply as a default:

```

entity nonFunctionalProperties
  dc:title
  dc:creator
  dc:subject
  dc:description
  dc:publisher
  dc:contributor
  dc:date
  dc:type
  dc:format
  dc:identifier
  dc:source
  dc:language
  dc:relation
  dc:coverage
  dc:rights
  version

```

Title

A name given to an element. Typically, `dc:title` will be a name by which the element is formally known.

Creator

An entity primarily responsible for creating the content of the element. Examples of `dc:creator` include a person, an organization, or a service. The Dublin Core specification recommends, that typically, the name of a `dc:creator` should be used to indicate the entity.

WSMO Recommendation: In order to point unambiguously to a specific resource we recommend the use an instance of `foaf:Agent` as value type [[Brickley & Miller, 2004](#)].

Subject

A topic of the content of the element. Typically, `dc:subject` will be expressed as keywords, key phrases or classification codes that describe a topic of the element. Recommended best practice is to select a value from a controlled vocabulary or formal classification scheme.

Description

An account of the content of the element. Examples of `dc:description` include, but are not limited to: an abstract, table of contents, reference to a graphical representation of content or a free-text account of the content.

Publisher

An entity responsible for making the element available. Examples of `dc:publisher` include a person, an organization, or a service. The Dublin Core specification recommends, that typically, the name of a `dc:publisher` should be used to indicate the entity.

WSMO Recommendation: In order to point unambiguously to a specific resource we recommend the use an instance of `foaf:Agent` as value type [[Brickley & Miller, 2004](#)].

Contributor

An entity responsible for making contributions to the content of the element. Examples of `dc:contributor` include a person, an organization, or a service. The Dublin Core specification recommends, that typically, the name of a `dc:contributor` should be used to indicate the entity.

WSMO Recommendation: In order to point unambiguously to a specific resource we recommend the use an instance of `foaf:Agent` as value type [[Brickley & Miller, 2004](#)].

Date

A date of an event in the life cycle of the element. Typically, `dc:date` will be associated with the creation or availability of the element.

WSMO Recommendation: We recommend to use the an encoding defined in the ISO Standard 8601:2000 [[ISO8601, 2004](#)] for date and time notation. A short introduction

on the standard can be found [here](#). This standard is also used by the the XML Schema Definition (YYYY-MM-DD) [[Biron & Malhotra, 2001](#)] and thus one is automatically compliant with XML Schema too.

Type

The nature or genre of the content of the element. The `dc:type` includes terms describing general categories, functions, genres, or aggregation levels for content.
WSMO Recommendation: We recommend to use an URI encoding to point to the namespace or document describing the type, e.g. for a domain ontology expressed in WSMO, one would use: <http://www.wsmo.org/2004/d2/#ontologies>.

Format

A physical or digital manifestation of the element. Typically, `dc:format` may include the media-type or dimensions of the element. Format may be used to identify the software, hardware, or other equipment needed to display or operate the element. Examples of dimensions include size and duration.

WSMO Recommendation: We recommend to use types defined in the list of Internet [Media Types](#) [[IANA, 2002](#)] by the IANA (Internet Assigned Numbers Authority)

Identifier

An unambiguous reference to the element within a given context. Recommended best practice is to identify the element by means of a string or number conforming to a formal identification system. In Dublin Core formal identification systems include but are not limited to the Uniform element Identifier (URI) (including the Uniform element Locator (URL)), the Digital Object Identifier (DOI) and the International Standard Book Number (ISBN).

WSMO Recommendation: We recommend to use URIs as Identifier, depending on the particular syntax the identity information of an element might already be given, however it might be repeated in `dc:identifier` in order to allow Dublin Core meta data aware applications the processing of that information.

Source

A reference to an element from which the present element is derived. The present element may be derived from the `dc:source` element in whole or in part.

Recommended best practice is to identify the referenced element by means of a string or number conforming to a formal identification system.

WSMO Recommendation: We recommend to use URIs as Identifier where possible.

Language

A language of the intellectual content of the element.

WSMO Recommendation: We recommend to use the language tags defined in the ISO Standard 639 [[ISO639, 1988](#)], e.g. "en-GB", in addition the logical language used to express the content should be mentioned, for example this can be [OWL](#).

Relation

A reference to a related element. Recommended best practice is to identify the referenced element by means of a string or number conforming to a formal identification system.

WSMO Recommendation: We recommend to use URIs as Identifier where possible. In particular, this property can be used to define namespaces that can be used in all child elements of the element to which this non functional property is assigned to.

Coverage

The extent or scope of the content of the element. Typically, `dc:coverage` will include spatial location (a place name or geographic coordinates), temporal period (a period label, date, or date range) or jurisdiction (such as a named administrative entity).

WSMO Recommendation: For more complex applications, consideration should be given to using an encoding scheme that supports appropriate specification of information, such as [DCMI Period](#), [DCMI Box](#) or [DCMI Point](#).

Rights

Information about rights held in and over the element. Typically, `dc:rights` will contain a rights management statement for the element, or reference a service providing such information. Rights information often encompasses Intellectual Property Rights (IPR), Copyright, and various Property Rights. If the Rights element

is absent, no assumptions may be made about any rights held in or over the element.

Version

As many properties of an element might change in time, an identifier of the element at a certain moment in time is needed.

WSMO Recommendation: If applicable we recommend to use the revision numbers of a version control system. Such a system can be for example CVS (Concurrent Version System), that automatically keeps track of the different revisions of a document, an example CVS version Tag looks like: "\$Revision: 1.66 \$".

2.5.2. Non functional properties - web service specific properties

Besides the core properties described in the previous section, the web service specific non functional properties also include properties related to the quality aspect of a web service (QoS):

Listing 2. Non functional properties definition for web services

```
entity wsNonFunctionalProperties subEntityOf nonFunctionalProperties
  accuracy
  availability
  financial
  networkRelatedQoS
  performance
  reliability
  robustness
  scalability
  security
  transactional
  trust
```

Accuracy

It represents the error rate generated by the web service. It can be measured by the numbers of errors generated in a certain time interval.

Availability

Refers to the temporal (i.e. when) and spatial (i.e. where) constraints applied to a service. In the first case, one could for instance specify that a service is only functional at certain periods of time (e.g. not at from 10pm to 2am every day or not at certain days during a year). In the second case, one could imagine a provider which sells goods only for customers in a certain area, like an electronic web service for ordering pizzas whose provider is located in New York and not interested in taking orders of people located in San Francisco. Availability is a complex property of services [O`Sullivan et al., 2002].

Financial

It represents the cost-related and charging-related properties of a web service [O`Sullivan et al., 2002]. This property is a complex property, which includes charging styles (e.g. per request or delivery, per unit of measure or granularity etc.), aspects of settlement like the settlement model (transactional vs. rental) and a settlement contract, payment obligations and payment instruments.

Network-related QoS

They represent the QoS mechanisms operating in the transport network which are independent of the web services. They can be measured by network delay, delay variation and/or message loss.

Performance

It represents how fast a service request can be completed. According to [Rajesh & Arulazi, 2003] performance can be measured in terms of throughput, latency, execution time, and transaction time. The response time of a service can also be a measure of the performance. High quality web services should provide higher

throughput, lower latency, lower execution time, faster transaction time and faster response time.

Reliability

It represents the ability of a web service to perform its functions (to maintain its service quality). It can be measured by the number of failures of the service in a certain time interval.

Robustness

It represents the ability of the service to function correctly in the presence of incomplete or invalid inputs. It can be measured by the number of incomplete or invalid inputs for which the service still function correctly.

Scalability

It represents the ability of the service to process more requests in a certain time interval. It can be measured by the number of solved requests in a certain time interval.

Security

It represents the ability of a service to provide authentication (entities - users or other services - who can access service and data should be authenticated), authorization (entities should be authorized so that they only can access the protected services), confidentiality (data should be treated properly so that only authorized entities can access or modify the data), traceability/auditability (it should be possible to trace the history of a service when a request was serviced), data encryption (data should be encrypted), and non-repudiation (an entity cannot deny requesting a service or data after the fact).

Transactional

It represents the transactional properties of the web service.

Trust

It represents the trust worthiness of the service.

Further discussions on service specific nonFunctionalProperties that can be used during the service life cycle can be found in [\[O` Sullivan et al., 2002\]](#). In conclusion, the web service specific non functional properties extend the common core properties (in [Section 2.4.1](#)) especially by quality of service aspects. Nonetheless, the model is extensible and more (even application-domain specific) aspects could be added.

3. Ontologies

In WSMO Ontologies are the key to link conceptual real world semantics defined and agreed upon by communities of users. *An ontology is a formal explicit specification of a shared conceptualization* [\[Gruber, 1993\]](#). From this rather conceptual definition we want to extract the essential components which define an ontology. Ontologies define a common agreed upon terminology by providing concepts and relationships among the set of concepts. In order to capture semantic properties of relations and concepts, an ontology generally also provides a set of axioms, which means expressions in some logical framework. An ontology is defined as follows:

Listing 3. Ontology definition

```
entity ontology
  nonFunctionalProperties ofType nonFunctionalProperties
  importOntologies ofTypeSet ontology
  usedMediators ofTypeSet ooMediator
  concepts ofTypeSet concept
  relations ofTypeSet relation
  functions ofTypeSet function
  instances ofTypeSet instance
  axioms ofTypeSet axiom
```

3.1 Non functional properties

The `nonFunctionalProperties` of an ontology consist of the core properties described in [Section 2.5.1](#).

3.2 Import Ontologies

Building an ontology for some particular problem domain can be a rather cumbersome and complex task. One standard way to deal with the complexity is modularization.

`ImportOntologies` allow a modular approach for ontology design; this simplified statement can be used as long as no conflicts need to be resolved, otherwise an `ooMediator` needs to be used.

3.3 Used mediators

When importing ontologies, most likely some steps for aligning, merging and transforming imported ontologies have to be performed. For this reason and in line with the basic design principles underlying the WSMF, ontology mediators (`ooMediator`) are used when an alignment of the imported ontology is necessary. Mediators are described in [Section 5](#) in more detail.

3.4 Concepts

Concepts constitute the basic elements of the agreed terminology for some problem domain. From a high level perspective, a concept – described by a concept definition – provides attributes with names and types. Furthermore, a concept can have several (possibly none) direct superconcepts as specified by the "isA"-relation.

Listing 4. Concept definition

```
entity concept
  nonFunctionalProperties ofType nonFunctionalProperties
  superConcepts ofTypeSet concept
  attributes ofTypeSet attribute
  definedBy ofType logicalExpression
```

Non functional properties

The `nonFunctionalProperties` of a concept consist of the core properties described in the [Section 2.5.1](#).

Superconcepts

There can be a finite number of concepts that serve as a `superConcepts` for some concept. Being a subconcept of some other concept in particular means that a concept inherits the signature of this superconcept and the corresponding constraints. Furthermore, all instances of a concept are also instances of its superconcept.

Attributes

Each concept provides a (possibly empty) set of `attributes` that represent named slots for data values for instances that have to be filled at the instance level. An attribute specifies a slot of a concept by fixing the name of the slot as well as a logical constraint on the possible values filling that slot. Hence, this logical expression can be interpreted as a typing constraint.

Listing 5. Attribute definition

```
entity attribute
  nonFunctionalProperties ofType nonFunctionalProperties
  range ofType concept
```

Non functional properties

The `nonFunctionalProperties` of an `attribute` consist of the core properties described in the Section 2.5.1.

Range

A `concept` that serves as an integrity constraint on the values of the attribute.

Defined by

A logical expression (see [Section 7](#)) which can be used to define the semantics of the concept formally. More precisely, the logical expression defines (or restricts, resp.) the extension (i.e. the set of instances) of the concept. If `C` is the identifier denoting the concept then the logical expression takes one of the following forms

- forall ?x (?x memberOf C -> l-expr(?x))
- forall ?x (?x memberOf C <- l-expr(?x))
- forall ?x (?x memberOf C <-> l-expr(?x))

where `l-expr(?x)` is a logical expression with precisely one free variable `?x`.

In the first case, one gives a necessary condition for membership in the extension of the concept; in the second case, one gives a sufficient condition and in the third case, we have a sufficient and necessary condition for an object being an element of the extension of the concept.

3.5 Relations

`Relations` are used in order to model interdependencies between several concepts (respectively instances of these concepts).

Listing 6. Relation definition

```
entity relation
  nonFunctionalProperties ofType nonFunctionalProperties
  superRelations ofTypeSet relation
  parameters ofTypeSet parameter
  definedBy ofType logicalExpression
```

Non functional properties

The `nonFunctionalProperties` of a relation consist of the core properties described in the Section 2.5.1.

Superrelations

A finite set of `relations` of which the defined relation is declared as being a subrelation. That particularly implies that the set of tuples belonging to the relation (the extension of the relation, resp.) is a subset of each of the extensions of the superrelations.

Parameters

A list of parameters; a parameter is a named placeholder for some value.

Listing 7. Parameter definition

```
entity parameter
  domain ofType concept
```

Non functional properties

The `nonFunctionalProperties` of a `parameter` consist of the core properties described in the Section 2.5.1.

Domain

A `concept` constraining the possible values that the parameter can take.

Defined by

A `logicalExpression` (see [Section 7](#)) defining the set of instances (n-ary tuples, if n is the arity of the relation) of the relation. The relation is represented by a n-ary predicate symbol with named arguments (see [Section 7](#)) (where n is the number of parameters of the relation) where the identifier of the relation is used as the name of the relation symbol.

If R is the identifier denoting the relation then the logical expression takes one of the following forms

- `forall ?v1, ..., ?vn (R(p1:?v1, ..., pn:?vn) -> l-expr(?v1, ..., ?vn))`
- `forall ?v1, ..., ?vn (R(p1:?v1, ..., pn:?vn) <- l-expr(?v1, ..., ?vn))`
- `forall ?v1, ..., ?vn (R(p1:?v1, ..., pn:?vn) <-> l-expr(?v1, ..., ?vn))`

where `l-expr(?v1, ..., ?vn)` is a logical expression with precisely `?v1, ..., ?vn` as its free variables and `p1, ..., pn` are the names of the parameters of the relation.

In the first case, one gives a necessary condition for instances `?v1, ..., ?vn` to be related; in the second case, one gives a sufficient condition and in the third case, we have a sufficient and necessary condition for instances `?v1, ..., ?vn` being related.

3.6 Functions

A `function` is a special relation, with a unary range and a n-ary domain (`parameters` inherited from `relation`), where the range specifies the return value. In contrast to a function symbol, a function is not only a syntactical entity but has some semantics that allows to actually evaluate the function if one considers concrete input values for the parameters of the function. That means, that we actually can replace the (ground) function term in some expression by its concrete value. Function can be used for instance to represent and exploit built-in predicates of common datatypes. Their semantics can be captured externally by means of an oracle or it can be formalized by assigning a logical expression to the `definedBy` property inherited from `relation`.

Listing 8. Function definition

```
entity function subEntityOf relation
  range ofType concept
```

Range

A `concept` constraining the possible return values of the function.

The logical representation of a function however is slightly different from relations, thus the logical expression used when defining the semantics of a function has to use a slightly different logical representation:

The function is represented by an (n)-ary function symbol with named arguments (see [Section 7](#)) (where n is the number of arguments of the function) where the identifier of the function is used as the name of the function. If F is the identifier denoting the function and `p1, ..., pn` is the set of parameters of the function then the logical expression can for example take the form `forall ?v1, ..., ?vn, ?res (F(p1:?v1, ..., pn:?vn) = ?res <-> l-expr(?v1, ..., ?vn, ?res))`, where `l-expr(?v1, ..., ?vn, ?res)` is a logical expression with precisely `?v1, ..., ?vn, ?res` as its free variables and `p1, ..., pn` are the names of the

parameters of the function.

3.7 Instances

Instances are either defined explicitly or by a link to an instance store, i.e., an external storage of instances and their values.

An explicit definition of instances of concepts is as follows:

Listing 9. Instance definition

```
entity instance
  nonFunctionalProperties ofType nonFunctionalProperties
  instanceOf ofType concept
  attributeValues ofTypeSet attributeValue
```

Non functional properties

The `nonFunctionalProperties` of an `instance` consist of the core properties described in the Section 2.5.1.

Instance of

The `concept` to which the instance belongs to.

Attribute values

The `attributeValues` for the single attributes defined in the concept. For each attribute defined for the `concept` this instance is assigned to there should be a corresponding attribute value.

Listing 10. Attribute value definition

```
entity attributeValue
  nonFunctionalProperties ofType nonFunctionalProperties
  value ofType {instance, URIReference, Literal, AnonymousId}
```

Non functional properties

The `nonFunctionalProperties` of an `attributeValue` consist of the core properties described in the Section 2.5.1.

Value

An instance, URI reference, literal or anonymous id representing the actual value of an instance for a specific attribute.

Instances of relations (with arity n) can be seen as n -tuples of instances of the concepts which are specified as the parameters of the relation. Thus we use the following definition for instances of relations:

Listing 11. Relation instance definition

```
entity relationInstance
  nonFunctionalProperties ofType nonFunctionalProperties
  instanceOf ofType relation
  relatedInstances ofTypeSet parameterValue
```

Non functional properties

The `nonFunctionalProperties` of a `relationInstance` consist of the core properties described in the Section 2.5.1.

Instance of

The `relation` this instance belongs to.

Related instances

A set of `parameterValues` specifying the single instances that are related according to this relation instance. The list of parameter values of the instance has to be compatible wrt. names and range constraints of that are specified in the corresponding relation.

A detailed discussion and a concrete proposal on how to integrate large sets of instance data in an ontology model can be found in [DIP Deliverable D2.2 \[Kiryakov et. al., 2004\]](#). Basically, the approach there is to integrate large sets of instances which are already existing on some storage devices by means of sending queries to external storage devices or oracles.

3.8 Axioms

An `axiom` is considered to be a logical expression together with its non functional properties.

Listing 13. Axiom definition

```
entity axiom
  nonFunctionalProperties ofType nonFunctionalProperties
  definedBy ofType logicalExpression
```

Non functional properties

The `nonFunctionalProperties` of an axiom consist of the core properties described in [Section 2.5.1](#).

Defined by

The actual statement captured by the axiom is defined by an formula in a logical language as described in [Section 7](#).

4. Goals

In this section, we introduce the notion of `goals` and define the elements that are used in the description of a goal. A Goal is defined as follows:

Listing 14. Goal definition

```
entity goal
  nonFunctionalProperties ofType nonFunctionalProperties
  importOntologies ofTypeSet ontology
  usedMediators ofTypeSet {ooMediator, ggMediator}
  postConditions ofTypeSet axiom
  effects ofTypeSet axiom
```

Non functional properties

The `nonFunctionalProperties` of a goal consist of the core properties described in the [Section 2.4.1](#).

Import Ontologies

It is used to import ontologies as long as no conflicts are needed to be resolved.

Used mediators

A goal can import ontologies using ontology mediators (`ooMediators`) when steps for aligning, merging and transforming imported ontologies are needed. A goal may be defined by reusing one or several already existing goals. This is achieved by using goal mediators (`ggMediators`). For a detailed account on mediators we refer to [Section 5](#).

PostConditions

`PostConditions` in WSMO describe the state of the information space that is desired.
Effects

`Effects` describe the state of the world that is desired.

5. Mediators

In this section, we introduce the notion of `mediators` and define the elements that are used in the description of a mediator.

We distinguish four different types of mediators :

- `ggMediators`: mediators that link two goals. This link represents the refinement of the source goal into the target goal.
- `ooMediators`: mediators that import ontologies and resolve possible representation mismatches between ontologies.
- `wgMediators`: mediators that link web service to goals. They explicitly may state the difference between the two entities and map different vocabularies (through the use of `ooMediators`).
- `wwMediators`: mediators linking two Web Services.

The `mediator` is defined as follows:

Listing 15. Mediators definition

```
entity mediator
  nonFunctionalProperties ofType nonFunctionalProperties
  importOntologies ofTypeSet ontology
  source ofTypeSet {ontology, goal, webService, mediator}
  target ofType {ontology, goal, webService, mediator}
  mediationService ofType {goal, webService, wwMediator}

entity ooMediator subEntityOf mediator
  source ofTypeSet {ontology, ooMediator}

entity ggMediator subEntityOf mediator
  usedMediators ofTypeSet ooMediator
  source ofTypeSet {goal, ggMediator}
  target ofType {goal, ggMediator}

entity wgMediator subEntityOf mediator
  usedMediators ofTypeSet ooMediator
  source ofType {webService, wgMediator}
  target ofType {goal, ggMediator}

entity wwMediator subEntityOf mediator
  usedMediators ofTypeSet ooMediator
  source ofType {webService, wwMediator}
  target ofType {webService, wwMediator}
```

Non functional properties

The non functional properties of a mediator consist of the core properties described in the [Section 2.5.1](#) (where, in this case, an element in the core properties is equivalent to a mediator). Besides these properties, and taking into account that a mediator uses a mediation service, the non functional properties of a mediator also include aspects related to the quality aspect of the mediation service (see [Section 2.5.2](#) for a description of these properties). The quality aspects of the mediator and the mediation service, taken together, might be enhanced (e.g. improving the robustness) or weakened (e.g. increasing the response time) by the mediator.

Import Ontologies

It is used to import ontologies as long as no conflicts are needed to be resolved.

Source

The source components define entities that are the sources of the mediator.

Target

The target component defines the entity that is the targets of the mediator.

Mediation Service

The `mediationService` points to a goal that declarative describes the mapping or to a web service that actually implements the mapping or to a `wwMediator` that links to a web service that actually implements the mapping.

Used Mediators

Some specific types of mediators, i.e. `ggMediator`, `wgMediator` and `wwMediator`, use a set of `ooMediators` in order to map between different vocabularies used in the description of goals and webservice capabilities and align different heterogeneous ontologies.

Notice that there are two principled ways of relating mediators with other entities in the WSMO model: (1) an entity can specify a relation with a mediator through the `usedMediators` attribute and (2) entities can be related with mediators through the source and target attributes of the mediator. We expect cases in which a mediator needs to be referenced directly from an entity, for example for importing a particular ontology necessary for the descriptions in the entity. We also expect cases in which not the definition of the entity itself, but rather the use of entities in a particular scenario (e.g. web service invocation) requires the use of mediators. In such a case, a mediator needs to be selected, which provides mediation services between these particular entities. WSMO does not prescribe the type of use of mediators and therefore provides maximal flexibility in the use of mediators and thus allows for loose coupling between web services, goals and ontologies.

6. Web Services

In this section we identify the concepts needed for describing various aspects of a web service. The following properties of a web service are considered: `namespaces`, `nonFunctionalProperties`, `importOntologies`, `usedMediators`, `capability` and `interfaces`.

Listing 16. Web service definition

```
entity webService
  nonFunctionalProperties ofType wsNonFunctionalProperties
  importOntologies ofTypeSet ontology
  usedMediators ofTypeSet ooMediator
  capability ofType capability
  interfaces ofTypeSet interface
```

Non functional properties

The `nonFunctionalProperties` of a web service are described in [Section 2.5.2](#).

Import Ontologies

It is used to import ontologies as long as no conflicts are needed to be resolved.

Used mediators

A web service can import ontologies using ontology mediators (`ooMediators`) when steps for aligning, merging and transforming imported ontologies are needed.

Capability

The `capability` of a web service is described in [Section 6.1](#).

Interfaces

The `interfaces` of a web service are described in [Section 6.2](#).

6.1 Capability

A `capability` defines the web service by means of its functionality.

Listing 17. Capability definition

```
entity capability
  nonFunctionalProperties ofType nonFunctionalProperties
  importOntologies ofTypeSet ontology
  usedMediators ofTypeSet {ooMediator, wgMediator}
  preconditions ofTypeSet axiom
  assumptions ofTypeSet axiom
  postconditions ofTypeSet axiom
  effects ofTypeSet axiom
```

Non functional properties

The `nonFunctionalProperties` of a capability consist of the core properties described in the [Section 2.5.1](#).

Import Ontologies

It is used to import ontologies as long as no conflicts are needed to be resolved.

Used mediators

A capability can import ontologies using ontology mediators (`ooMediators`) when steps for aligning, merging and transforming imported ontologies are needed. It can be linked to a goal using a `wgMediator`.

PreConditions

`PreConditions` in WSMO describe what a web service expects for enabling it to provide its service. In other words, they constrain the set of states of the information space such that each state satisfying these constraints can serve as a valid starting state (in the information space) for executing the service in a defined manner.

Assumptions

`Assumptions` in WSMO describe the expectation of the service on the state of the world when starting an execution of the service. The service guarantees the declared functionality only if it is started in such a state. Thus, the assumptions constrain the set of states of the world to the set of valid starting states.

PostConditions

`PostConditions` in WSMO describe the states of the information space that must be reached by executing the service.

Effects

`Effects` describe the state of the world that must to be reached by executing the service.

6.2 Interfaces

An `interface` describes how the functionality of the service can be achieved (i.e. how the `capability` of a service can be fulfilled) by providing a twofold view on the operational competence of the service:

- `choreography` decomposes a capability in terms of interaction with the service (service user's view)
- `orchestration` decomposes a capability in terms of functionality required from other services (other service providers' view)

This distinction reflects the difference between communication and cooperation. The `choreography` defines how to communicate with the web service in order to consume its functionality. The `orchestration` defines how the overall functionality is achieved by the cooperation of more elementary service providers.

An `interface` is defined by the following properties:

Listing 18. Interface definition

```
entity interface
  nonFunctionalProperties ofType nonFunctionalProperties
  importOntologies ofTypeSet ontology
  usedMediators ofTypeSet ooMediator
  choreography ofType choreography
  orchestration ofType orchestration
```

Non functional parameters

The `nonFunctionalProperties` of an interface consist of the core properties described in the [Section 2.5.1](#).

Import Ontologies

It is used to import ontologies as long as no conflicts are needed to be resolved.

Used mediators

An interface can import ontologies using ontology mediators (`ooMediators`) when steps for aligning, merging and transforming imported ontologies are needed.

Choreography

`Choreography` provides the necessary information for the user to communicate with the web service (i.e. it describes how the service works and how to access the service from the user's perspective).

Orchestration

`Orchestration` describes how the service works from the provider's perspective (i.e. how a service makes use of other web service or goals in order to achieve its capability).

The distinction between `choreography` and `orchestration` [1] should be considered in the context of the role the service is playing in a conversation: provider or requester. In case the service acts as a provider, the way of interacting with it is specified in its choreography. If the service acts as a requester, requesting functionalities of different services, then

- the way in which this services are composed and
- the way of interacting with them

are specified in the orchestration of the service.

7. Logical language for defining formal statements in WSMO

As the major component of `axiom`, logical expressions are used almost everywhere in the WSMO model to capture specific nuances of meaning of modeling elements or their constituent parts in a formal and unambiguous way. In the following, we give a definition of the syntax of the formal language that is used for specifying `logicalExpressions`. The semantics of this language will be defined formally by the WSMO working group in a separate document.

The language defines here basically is a first-order language, similar to First-order Logics [Enderton, 1972] and Frame Logic (F-Logic, resp.) [Kifer et al., 1995]. In particular, we exploit the advanced object-oriented modeling constructs of F-Logic and reflect these constructs in our language.

We start with the definition of the basic vocabulary for building logical expression. Then we define the set of terms and the most basic formulas (atomic formulae, resp.) which allows us to eventually define the set of logical expressions.

Let *URI* be the set of all valid uniform resource identifiers. This set will be used for the naming (or identifying, resp.) various entities in a WSMO description.

Definition 1. The **vocabulary V** of our language $L(V)$ consists of the following symbols:

- The set of all **Uniform Resource Identifiers** *URI*.
- The set of all **QNames** *QN*.
- The set of all **anonymous Ids** *AnID*.
- The set of all **literals** *Lit*.
- An infinite set of **variables** *Var*.
- An infinite set of **function symbols** (object constructors, resp.) *FSym* which is a subset of *URI*.
- An infinite set of **function symbols with named arguments** *FSymNamed* which is a subset of *URI*.
- An infinite set of **predicate symbols** *PSym* which is a subset of *URI*.
- An infinite set of **predicate symbols with named arguments** *PSymNamed* which is a subset of *URI*.
- A finite set of **auxiliary symbols** *AuxSym* including $(,)$, *ofType*, *ofTypeSet*, *memberOf*, *subConceptOf*, *hasValue*, *hasValues*, *false*, *true*.
- A finite set of **logical connectives and quantifiers** including the usual ones from First-Order Logics: *or*, *and*, *not*, \leftarrow , \rightarrow , \leftrightarrow , *forall*, *exists*.
- All these sets are assumed to be *mutually distinct*.
- For each symbol *S* in *FSym*, *FSymNamed*, *PSym* or *PSymNamed*, we assume that there is a corresponding **arity** *arity(S)* defined, which is a non-negative integer specifying the number of arguments that are expected by the corresponding symbol when building expressions in our language.
- For each symbol *S* in *FSymNamed* or *PSymNamed*, we assume that there is a corresponding **set of parameter names** *parNames(S)* defined, which gives the names of the single parameters of the symbol that have to be used when building expressions in our language using these symbols.

As usual, 0-ary function symbols are called *constants*. 0-ary predicate symbols correspond to propositional variables in classical propositional logic.

Definition 2. Given a vocabulary V , we can define the **set of terms** $Term(V)$ (over vocabulary V) as follows:

- Any identifier u in *URI* is a term in $Term(V)$.
- Any QName q in *QN* is a term in $Term(V)$.
- Any anonymous Id i in *AnID* is a term in $Term(V)$.
- Any literal l in *Lit* is a term in $Term(V)$.
- Any variable v in *Var* is a term in $Term(V)$.
- If f is a function symbol from *FSym* with $arity(f) = n$ and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term in $Term(V)$.
- If f is a function symbol (with named arguments) from *FSymNamed* with $arity(f) = n$, $parNames(f) = \{p_1, \dots, p_n\}$ and t_1, \dots, t_n are terms, then $f(p_1:t_1, \dots, p_n:t_n)$ is a term in $Term(V)$.
- Nothing else is a term.

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

Terms can be used in general to describe computations (in some domain). One important additional interpretation of terms is that they denote objects in some universe and thus provide names for entities in some domain of discourse.

Definition 3. We extend this definition to the **set of logical expression** (or formulae,

resp.) $L(V)$ (over vocabulary V) as follows:

A simple `logicalExpression` in $L(V)$ (or atomic formula) is inductively defined by

- If p is a predicate symbol in $PSym$ with $arity(p) = n$ and t_1, \dots, t_n are terms, then $p(t_1, \dots, t_n)$ is a simple logical expression in $L(V)$.
- If r is a predicate symbol with named arguments in $PSymNamed$ with $arity(p) = n$, $parNames(r) = \{p_1, \dots, p_n\}$ and t_1, \dots, t_n are terms, then $r(p_1:t_1, \dots, p_n:t_n)$ is a simple logical expression in $L(V)$.
- `true` and `false` are simple logical expression in $L(V)$.
- If P, ATT, T are terms in $Term(V)$, then $P[ATT\ of\ Type\ T]$ is a simple logical expression in $L(V)$.
- If P, ATT, T_1, \dots, T_n (where $n \geq 1$) are terms in $Term(V)$, then $P[ATT\ of\ Type\ Set\ (T_1, \dots, T_n)]$ is a simple logical expression in $L(V)$.
- If O, T are terms in $Term(V)$, then $O\ memberOf\ T$ is a simple logical expression in $L(V)$.
- If C_1, C_2 are terms in $Term(V)$, then $C_1\ subConceptOf\ C_2$ is a simple logical expression in $L(V)$.
- If O, V, ATT are terms in $Term(V)$, then $O[ATT\ has\ Value\ V]$ is a simple logical expression in $L(V)$.
- If O, V_1, \dots, V_n, ATT (where $n \geq 1$) are terms in $Term(V)$, then $O[ATT\ has\ Values\ \{V_1, \dots, V_n\}]$ is a simple logical expression in $L(V)$.
- If T_1 and T_2 are terms in $Term(V)$, then $T_1 = T_2$ is a simple logical expression in $L(V)$.
- Nothing else is a simple logical expression.

The intuitive semantics for simple logical expressions (wrt. an interpretation) is as follows:

- The semantics of predicates in $PSym$ is the common one for predicates in First-Order Logics, i.e. they denote basic statements about the elements of some universe which are represented by the arguments of the symbol.
- Predicates with named arguments have the same semantic purpose but instead of identifying the arguments of the predicate by means a fixed order, the single arguments are identified by a parameter name. The order of the arguments does not matter here for the semantics of the predicate but the corresponding parameter names. Obviously, this has consequences for unification algorithms.
- `true` and `false` denote atomic statements which are always true (or false, resp.)
- $C[ATT\ of\ Type\ T]$ defines a constraint on the possible values that instances of class C may take for property ATT to values of type T . Thus, this is expression is a signature expression.
- The same purpose has the simple logical expression $C[ATT\ of\ Type\ Set\ (T_1, \dots, T_n)]$. It defines a constraint on the possible values that instances of class C may take for property ATT to values of types T_1, \dots, T_n . That means all values of all the specified types are allowed as values for the property ATT .
- $O\ memberOf\ T$ is true, iff element o is an instance of type T , that means the element denoted by o is a member of the extension of type T .
- $C_1\ subConceptOf\ C_2$ is true iff concept C_1 is a subconcept of concept C_2 , that means the extension of concept C_1 is a subset of the extension of concept C_2 .
- $O[ATT\ has\ Value\ V]$ is true if the element denoted by o takes value v under property ATT .
- Similar for the simple logical expression $O[ATT\ has\ Values\ \{V_1, \dots, V_n\}]$: The expression holds if the set of values that the element o takes for property ATT includes all the values v_1, \dots, v_n . That means the set of values of o for property ATT is a superset of the set $\{V_1, \dots, V_n\}$.
- $T_1 = T_2$ is true, if both terms T_1 and T_2 denote the same element of the universe.

Definition 4. This definition is extended to complex `logicalExpression` in $L(V)$ as follows

- Every simple logical expression in $L(V)$ is a logical expression in $L(V)$.
- If L is a logical expression in $L(V)$, then `not L` is a logical expression in $L(V)$.

- If L_1 and L_2 are logical expressions in $L(V)$ and op is one of the logical connectives in $\{or, and, implies, equivalent, <- \}$, then $L_1 op L_2$ is a logical expression in $L(V)$.
- If L is a logical expression in $L(V)$, x is a variable from Var and Q is a quantor in $\{forall, exists \}$, then $Qx(L)$ is a logical expression in $L(V)$.
- Nothing else is a logical expression (or formula, resp.) in $L(V)$.

The intuitive semantics for complex logical expressions (wrt. to in interpretation) is as follows:

- not L is true iff the logical expression L does not hold
- $or, and, implies, equivalent, <-$ denote the common disjunction, conjunction, implication, equivalence and backward implication of logical expressions
- $forall x (L)$ is true iff L holds for all possible assignments of x with an element of the universe.
- $exists x (L)$ is true iff there is an assignment of x with an element of the universe such that L holds.

Notational conventions:

There is a precedence order defined for the logical connectives as follows, where $op_1 < op_2$ means that op_2 binds stronger than op_1 : $implies, equivalent, <- < or, and < not$.

The precedence order can be exploited when writing logical expressions in order to prevent from extensive use of parenthesis. In case that there are ambiguities in evaluating an expression, parenthesis must be used to resolve the ambiguities.

The terms $O[ATT\ ofTypeSet\ (T)]$ and $O[ATThasValues\ \{V\}]$ (that means for the case $n = 1$ in the respective clauses above) can be written simpler by omitting the parenthesis.

A logical expression of the form $false <- L$ can be written using the following syntactical shortcut (commonly used in Logic Programming system for defining integrity constraints):
 $<- L$.

Furthermore, we allow path expressions as a syntactical shortcut for navigation related expressions: $p.q$ stands for the element which can be reached by navigating from p via property q . The property q has to be a non-set-valued property ($hasValue$). For navigation over set-valued properties ($hasValues$), we use a different expression $p..q$. Such path expressions can be used like a term wherever a term is expected in a logical expression.

Note: Note that this definition for our language $L(V)$ is extensible by extending the basic vocabulary V . In this way, the language for expressing logical expressions can be customized to the needs of some application domain.

Semantically, the various modeling elements of ontologies can be represented as follows: concepts can be represented as terms, relations as predicates with named arguments, functions as functions with named arguments, instances as terms and variables as variables.

8. Conclusions and further directions

This document presented the Web Service Modeling Ontology (WSMO) for describing several aspects related to web services, by refining the Web Service Modeling Framework (WSMF). The definition of the missing elements (choreography and orchestration) will be provided in separate deliverables of the [WSMO working group](#).

References

- [Arroyo & Stollberg, 2004]** S. Arroyo and M. Stollberg (Eds.): *WSMO Primer*, WSMO Deliverable D3.1, DERI Working Draft, 2004, latest version available at <http://www.wsmo.org/2004/d3/d3.1/>.
- [Baader et al., 2003]** F. Baader, D. Calvanese, and D. McGuinness: *The Description Logic Handbook*, Cambridge University Press, 2003.
- [Berners-Lee et al., 1998]** T. Berners-Lee, R. Fielding, and L. Masinter: *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax*, IETF, August 1998, available at <http://www.isi.edu/in-notes/rfc2396.txt>.
- [Biron & Malhotra, 2001]** P. V. Biron and A. Malhotra: *XML Schema Part 2: Datatypes*, W3C Recommendation 02, 2001, available at: <http://www.w3.org/TR/xmlschema-2/>.
- [Bray et al., 1999]** T. Bray, D. Hollander, and A. Layman (Eds.): *Namespaces in XML*, W3C Recommendation REC-xml-names-19990114, 1999, available at: <http://www.w3.org/TR/REC-xml-names/>.
- [Brickley & Miller, 2004]** D. Brickley and L. Miller: *FOAF Vocabulary Specification*, available at: <http://xmlns.com/foaf/0.1/>.
- [Dean et al., 2004]** M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein: *OWL Web Ontology Language Reference*, W3C Recommendation, 10 February 2004. Available at <http://www.w3.org/TR/2004/REC-owl-ref-20040210/>.
- [Enderton, 1972]** H.B. Enderton: *A Mathematical Introduction to Logic*, Academic Press, 1972.
- [Fensel & Bussler, 2002]** D. Fensel and C. Bussler: *The Web Service Modeling Framework WSMF*, Electronic Commerce Research and Applications, 1(2), 2002.
- [Gruber, 1993]** T. Gruber: A translation approach to portable ontology specifications, *Knowledge Acquisition*, 5:199-220, 1993.
- [Hayes, 2004]** P. Hayes (ed.): *RDF Semantics*, W3C Recommendation 10 February 2004, 2004.
- [IANA, 2002]** Internet Assigned Number Authority: *MIME Media Types*, available at: <http://www.iana.org/assignments/media-types/>, February 2002.
- [ISO639, 1988]** International Organization for Standardization (ISO): *ISO 639:1988 (E/F). Code for the Representation of Names of Languages*. First edition, 1988-04-01. Reference number: ISO 639:1988 (E/F). Geneva: International Organization for Standardization, 1988. iii + 17 pages.
- [ISO8601, 2004]** International Organization for Standardization (ISO): *ISO 8601:2000. Representation of dates and times*. Second edition, 2004-06-08. Reference number. Geneva: International Organization for Standardization, 2004. Available from <http://www.iso.ch>.
- [Kifer et al., 1995]** M. Kifer, G. Lausen, and J. Wu: *Logical foundations of object-oriented and frame-based languages*. *Journal of the ACM*, 42:741-843, July 1995.
- [Kiryakov et al., 2004]** A. Kiryakov, D. Ognyanov, and V. Kirov: *A framework for representing ontologies consisting of several thousand concepts definitions*, Project Deliverable D2.2 of DIP, June 2004.

[Manoler & Miller, 2004] F. Manola and E. Miller: *RDF Primer*, W3C Working Draft 23 July 2004, available at <http://www.w3.org/TR/xpath-functions/>.

[Malhotra et al., 2004] A. Malhotra, J. Melton, N. Walsh: *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Recommendation 10 February 2004, available at <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>.

[O`Sullivan et al., 2002] J. O`Sullivan, D. Edmond, and A. Ter Hofstede: *What is a Service?: Towards Accurate Description of Non-Functional Properties*, Distributed and Parallel Databases, 12:117-133, 2002.

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

[Parr & Quong, 1995] Parr, T.J. and R.W. Quong: *ANTLR: A predicated-LL(k) parser generator*. Software, Practice and Experience, **25**(7), pp. 789-810, Jul 1995.

[Rajesh & Arulazi, 2003] S. Rajesh and D. Arulazi: *Quality of Service for Web Services-Demystification, Limitations, and Best Practices*, March 2003. (See <http://www.developer.com/services/article.php/2027911>.)

[Stollberg et al., 2004] M. Stollberg, H. Lausen, A. Polleres, and R. Lara (Eds.): *WSMO Use Case Modeling and Testing*, WSMO Deliverable D3.2, DERI Working Draft, 2004, latest version available at <http://www.wsmo.org/2004/d3/d3.2/>

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

[Yang & Kifer, 2003] G. Yang and M. Kifer: *Reasoning about Anonymous Resources and Meta Statements on the Semantic Web* J. Data Semantics I 2003: 69-97.

Acknowledgement

The work is funded by the European Commission under the projects [DIP](#), [Knowledge Web](#), [SEKT](#), [SWWS](#), and [Esperanto](#); by [Science Foundation Ireland](#) under the [DERI-Lion](#) project; and by the Vienna city government under the [CoOperate](#) program.

The editors would like to thank to all the members of the [WSMO](#), [WSML](#), and [WSMX](#) working groups for their advice and input into this document.

Appendix A. Conceptual Elements of WSMO

entity nonFunctionalProperties

- dc:title
- dc:creator
- dc:subject
- dc:description
- dc:publisher
- dc:contributor
- dc:date
- dc:type
- dc:format
- dc:identifier
- dc:source
- dc:language
- dc:relation

dc:coverage
dc:rights
version

entity wsNonFunctionalProperties

performance
reliability
security
scalability
robustness
accuracy
transactional
trust
financial
networkRelatedQoS

entity ontology

nonFunctionalProperties **ofType** nonFunctionalProperties
importOntologies **ofTypeSet** ontology
usedMediators **ofTypeSet** ooMediator
concepts **ofTypeSet** concept
relations **ofTypeSet** relation
functions **ofTypeSet** function
instances **ofTypeSet** instance
axioms **ofTypeSet** axiom

entity concept

nonFunctionalProperties **ofType** nonFunctionalProperties
superConcepts **ofTypeSet** concept
attributes **ofTypeSet** attribute
definedBy **ofType** logicalExpression

entity goal

nonFunctionalProperties **ofType** nonFunctionalProperties
importOntologies **ofTypeSet** ontology
usedMediators **ofTypeSet** (ooMediator or ggMediator)
postConditions **ofType** goalAxiomDefinition
effects **ofType** goalAxiomDefinition

entity attribute

nonFunctionalProperties **ofType** nonFunctionalProperties
range **ofType** concept

entity relation

nonFunctionalProperties **ofType** nonFunctionalProperties
superRelations **ofTypeSet** relation
parameters **ofTypeSet** parameter
definedBy **ofType** logicalExpression

entity parameter

domain **ofType** concept

entity function **subEntityOf** relation

range **ofType** concept

entity instance

nonFunctionalProperties **ofType** nonFunctionalProperties
instanceOf **ofType** concept
attributeValues **ofTypeSet** attributeValue

entity attributeValue
nonFunctionalProperties **ofType** nonFunctionalProperties
value **ofType** {instance, URIReference, Literal, AnonymousId}

entity relationInstance
nonFunctionalProperties **ofType** nonFunctionalProperties
instanceOf **ofType** relation
relatedInstances **ofTypeSet** {instance, URIReference, Literal, AnonymousId}

entity axiom
nonFunctionalProperties **ofType** nonFunctionalProperties
definedBy **ofType** logicalExpression

entity mediator
nonFunctionalProperties **ofType** nonFunctionalProperties
importOntologies **ofTypeSet** ontology
source **ofTypeSet** {ontology, goal, webService, mediator}
target **ofType** {ontology, goal, webService, mediator}
mediationService **ofType** {goal, webService, wwMediator}

entity ooMediator **subEntityOf** mediator
source **ofTypeSet** {ontology, ooMediator}

entity ggMediator **subEntityOf** mediator
usedMediators **ofTypeSet** ooMediator
source **ofTypeSet** {goal, ggMediator}
target **ofType** {goal, ggMediator}

entity wgMediator **subEntityOf** mediator
usedMediators **ofTypeSet** ooMediator
source **ofType** {webService, wgMediator}
target **ofType** {goal, ggMediator}

entity wwMediator **subEntityOf** mediator
usedMediators **ofTypeSet** ooMediator
source **ofType** {webService, wwMediator}
target **ofType** {webService, wwMediator}

entity webService
nonFunctionalProperties **ofType** wsNonFunctionalProperties
importOntologies **ofTypeSet** ontology
usedMediators **ofTypeSet** ooMediator
capability **ofType** capability
interfaces **ofTypeSet** interface

entity capability
nonFunctionalProperties **ofType** nonFunctionalProperties
importOntologies **ofTypeSet** ontology
usedMediators **ofTypeSet** {ooMediator, wgMediator}
preconditions **ofTypeSet** axiomDefinition
assumptions **ofTypeSet** axiomDefinition
postconditions **ofTypeSet** axiomDefinition
effects **ofTypeSet** axiomDefinition

entity interface
nonFunctionalProperties **ofType** nonFunctionalProperties
importOntologies **ofTypeSet** ontology
usedMediators **ofTypeSet** ooMediator
choreography **ofType** choreography
orchestration **ofType** orchestration

Appendix B. BNF Grammar for WSML

This Appendix describes the grammar for the Web Services Modeling Language (WSML). This language is meant for representing the concepts introduced in the Web Service Modeling Ontology (WSMO) in a human-readable way.

The language to write down this syntax is a variant of Extended Backus Nauer Form; a variant readable by SableCC. SableCC is a compiler compiler - a tool that can construct recognisers, parsers and compilers from a grammar specification. We have previously experimented with a different compiler compiler, namely ANTLR [Parr & Quong 1995]. However, SableCC can generate compilers for LALR(1) grammars - a superset of the LL(k) grammars that ANTLR can handle. For instance, LALR(1) grammars can be left recursive, which LL(k) languages cannot. Visit www.sablecc.org for more information.

```
// SableCC grammar for WSML
// (c) DERI - Eyal Oren
// $Revision: 1.66 $
// $Author: titi $
// GRAMMAR IS PRELIMINARY
// when D2 is stable, the grammar will be updated accordingly

//TODO: frame based syntax
//TODO: literal datatyping - true/false
//TODO: restrict logical-concept definitions
Package deri.wsml.validator;
Helpers
  all = [0 .. 0xffff];
  digit = ['0' .. '9'];
  uletter = ['A' .. 'Z'];
  lletter = ['a' .. 'z'];
  letter = uletter | lletter | '_';
  alphanum = digit | letter;

  most_uri_chars =
  // RFC 2396 unreserved
    '!' | '$' | '%' | '&' | '*' | '+' | '-' | '.' | ':' | ';' | '=' | '@' | '[' | '\]' | '^' | '_' | '`' | '{' | '~' | ' ' | '\t' | '\n' | '\r' | '\f' | '\l';
  // reserved
    '!' | '$' | '%' | '&' | '*' | '+' | '-' | '.' | ':' | ';' | '=' | '@' | '[' | '\]' | '^' | '_' | '`' | '{' | '~' | ' ' | '\t' | '\n' | '\r' | '\f' | '\l';
  // unwise
    '{' | '}' | '[' | '\]' | '^' | '_' | '`' | '{' | '~' | ' ' | '\t' | '\n' | '\r' | '\f' | '\l';
  // Delims: Escape and ref
    '%' | '#' | '"';
  urichar = alphanum | most_uri_chars | '(' | ')' | ',' | '*';
  nsbegin = alphanum | '_';
  nsrest = nsbegin | '-';
  nsname = nsbegin nsrest*;
  localchar = alphanum | most_uri_chars;
  lname = localchar+;

  tab = 9;
  cr = 13;
  lf = 10;
  eol = cr lf | cr | lf;

  //not_star = [all - ['*+ /']];
  not_cr_lf = [all - [cr + lf]];
  quote = '"';
  not_quote = [all - quote];

  not_star = [all - '*'];
  not_star_slash = [not_star - '/'];
  long_comment = /* not_star * '*' + ( not_star_slash not_star * '*' + ) * '/';
  begin_comment = /* | 'comment';
  short_comment = begin_comment not_cr_lf* eol;
  //long_comment = /* not_star * */;
  comment = short_comment | long_comment;

  blank = (' ' | tab | eol)+;
  qmark = '?';
  colon = ':';
  lpar = '(';
```

```

r_arrow = '->';
l_arrow = '<-';
d_arrow = '<->';
d_r_arrow = '->>';

Tokens
literal = quote not_quote* quote;
comma = ',';
point = '.';

// logic
gt = 'gt';
lt = 'lt';
gte = 'ge';
lte = 'le';
lpar = 'lpar';
rpar = 'rpar';
equals = '=';

and = 'and' | '&';
or = 'or' | '|';
in = 'in';
implies = 'implies' | r_arrow;
l_arrow = l_arrow;
equivalent = 'equivalent' | d_arrow;
not = 'not' | '~';

exists = 'exists';
forall = 'forall';

// wsml keywords
t_namespace = 'namespace';
t_targetnamespace = 'targetNamespace';

t_importontology = 'importOntology';
t_usemediator = 'useMediator';
t_oomediator = 'ooMediator';
t_ggmediator = 'ggMediator';
t_wgmediator = 'wgMediator';
t_wwmediator = 'wwMediator';
t_source = 'source';
t_target = 'target';
t_useservice = 'useService';
t_reduction = 'reduction';

t_goal = 'goal';
t_webservice = 'webservice';

t_use_capability = 'useCapability';
t_use_interface = 'useInterface';
t_capability = 'capability';
t_precondition = 'precondition';
t_postcondition = 'postcondition';
t_assumption = 'assumption';
t_effect = 'effect';
t_interface = 'interface';

t_choreography = 'choreography';
t_orchestration = 'orchestration';
t_placeholder = '****';

t_ontology = 'ontology';
t_concept = 'concept';
t_subconcept = 'subConceptOf' | '<<';
t_oftype = 'ofType';
t_set = 'set';
t_instance = 'instance';
t_memberof = 'memberOf' | '::';
t_hasvalue = 'hasValue' | r_arrow;
t_hasvalues = 'hasValues' | d_r_arrow;

t_relation = 'relation';
t_subrelation = 'subRelationOf';
t_parameter = 'parameter';
t_function = 'function';
t_variable_def = 'variable';
t_method = 'method';
t_range = 'range';

```

```

t_axiom = 'axiom';
t_ref_axiom = 'useAxiom';
t_beginlogic = 'logicalExpression';
t_version = 'version';

t_nfp = 'nonFunctionalProperties' | 'nfp';
t_endnfp = 'endNonFunctionalProperties' | 'endnfp';
t_definedby = 'definedBy';

prefix = nsname+ colon;
uri = '<' urichar+ '>';
qname = nsname+ | nsname colon lname;
t_variable = qmark alphanum+;
anonymous = '_#' digit*;

blank = blank;
comment = comment;

```

Ignored Tokens

```

blank,
comment;

```

Productions

```

wsml = namespace? definition*;

namespace = t_namespace uri? prefixdefinition* targetdefinition?;
targetdefinition = t_targetnamespace uri;
prefixdefinition = prefix uri;

definition = {goal} goal | {ontology} ontology | {webservice} webservice | {mediator} mediator ;

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

omediator =
  t_omediator id
  namespace?
  nfp?
  import_ontology?
  source*
  target*
  use_service?;

ggmediator =
  t_ggmediator id
  header*
  source*
  target*
  reduction?;

wgmediator =
  t_wgmediator id
  header*
  source*
  target*
  reduction?;

wwmediator =
  t_wwmediator id
  header*
  source*
  target*;

use_service = t_useservice id;
reduction = t_reduction axiom;
source = t_source id;
target = t_target id;

goal =
  t_goal id
  header*
  postcondition_or_effect+;

use_mediator = t_usemediator idlist;
import_ontology = t_importontology idlist;
postcondition_or_effect = {postcondition} t_postcondition axiom | {effect} t_effect axiom;

webservice =
  t_webservice id
  header*

```

```

    capability?
    interface*;

capability = {use_capability} t_use_capability id | {defined_capability} capabilitydef;

capabilitydef =
    t_capability id
    nfp?
    use_mediator?
    import_ontology?
    pre_post_ass_or_eff*;

pre_post_ass_or_eff =
    {precondition} t_precondition axiom |
    {assumption} t_assumption axiom |
    {post_or_effect} postcondition_or_effect;

interface = {use_interface} t_use_interface id | {defined_interface} interfacedef;
interfacedef =
    t_interface id
    nfp?
    use_mediator?
    import_ontology?
    choreography?
    orchestration*;

choreography = t_choreography t_placeholder;
orchestration = t_orchestration t_placeholder;

ontology =
    t_ontology id
    header*
    ontology_elements*
    ;

ontology_elements = {concept} concept | {axiom} axiom | {instance} instance | {relation} relation | {variable} variable | {

concept =
    t_concept id
    superconcept*
    nfp?
    attribute*
    log_definition?;
superconcept =
    {concept} t_subconcept idlist ; //| {logical_concept} t_superconcept log_expr;
attribute =
    {attribute} [attr]:id t_oftype t_set? [type]:id;
log_definition =
    t_definedby log_expr;
instance =
    t_instance [instance]:id
    t_memberof [type]:id
    nfp?
    attributevalue*
    log_definition?;

attributevalue = {single_value} [attr]:id t_hasvalue [value]:id | {set_value} [attr]:id t_hasvalues [values]:idlist;

relation =
    t_relation id
    superrelation*
    nfp?
    parameter*;
superrelation = {relation} t_subrelation idlist;
parameter = [parameter]:id t_oftype [type]:id;

function =
    t_function id
    superrelation*
    nfp?
    parameter*
    range;
range = t_range t_oftype id;

varlist = {one} t_variable | {more} t_variable comma varlist;
variable = {untypedvar} t_variable_def [untypedlist]:varlist | {typedvar} t_variable_def [typedlist]:varlist t_memberof id;

nfp =
    t_nfp

```

```

    nfpitem*
    t_endnfp;
hasvalue = {one} t_hasvalue | {more} t_hasvalues;
nfpitem = {nfp} id hasvalue idlist | {version} t_version hasvalue idlist; // nfp identifier (or version) followed by one or mo

axiom = {use_axiom} t_axiom id | {defined_axiom} t_axiom id nfp? log_definition;

log_expr = expr point;
expr = or_val | {imply} expr imply_op or_val;
or_val = and_val | {or} or_val or_op and_val;
and_val = comp_val | {and} and_val and_op comp_val;
comp_val = oo_val | {comp} comp_val comp_op oo_val;
oo_val = simple | {oo} oo_val oo_op simple;
simple =
    {term} term |
    {oftype} [instance]:term t_oftype [type]:term |
    {par} lpar expr rpar |
    {not} not simple |
    {quantified} quantified;
quantified = quantifier t_variable simple;

// function names and constant names are id's
// e.g. priceOfCar, PriceOfCar, car:price, double(), double(price), math:double(car:price)
term =
    {constant} id |
    {function} id lpar rpar |
    {anyargs} id lpar term termlist* rpar;
termlist = comma term;

id = {uri} uri | {qname} qname | {literal} literal | {anonymous} anonymous | {var} t_variable;
idlist = {id} id | {idlist} id comma idlist;

quantifier = {forall} forall | {exists} exists;
and_op = and ;
or_op = or;
comp_op = {gt} gt | {lt} lt | {gte} gte | {lte} lte | {equal} equals;
imply_op = {l_imply} l_arrow | {imply} implies | {equiv} equivalent;
oo_op = {member} t_memberof | {subconcept} t_subconcept | {value} t_hasvalue | {setvalue} t_hasvalues;

header =
{namespace} namespace |
{nfp} nfp |
{use_mediator} use_mediator |
{import_ontology} import_ontology;

```

[1] One could argue that `orchestration` should not be part of a public interface because it refers to how a service is implemented. However, this is a short-term view that does not reflect the nature of fully open and flexible eCommerce. Here companies shrink to their core processes were they are really profitable in. All other processes are sourced out and consumed as eServices. They advertise their services in their capability and choreography description and they advertise their needs in the orchestration interfaces. This enables on-the-fly creation of virtual enterprises in reaction to demands from the market place. Even in the dinosaurian time of eCommerce where large companies still exist, `orchestration` may be an important aspect. The `orchestration` of a service may not be made public but may be visible to the different departments of a large organization that compete for delivering parts of the overall service. Notice that the actual business intelligence of a service provider is still hidden. It is his capability to provide a certain functionality with a chorography that is very different from the sub services and their orchestration. The ability for a certain type of process management (the overall functionality is decomposed differently in the `choreography` and the `orchestration`) is were it comes in as a Silver bullet in the process. How he manages the difference between the process decomposition at the `choreography` and the `orchestration` level is the business intelligence of the web service provider.



[webmaster](#)

