D2v1.2. Web Service Modeling Ontology (WSMO)

WSMO Final Draft 13 April 2005

Final version:
http://www.wsmo.org/TR/d2/v1.2/20050413/
Latest version:
http://www.wsmo.org/TR/d2/v1.2/
Previous version:
http://www.wsmo.org/TR/d2/v1.2/20050402/
Editors:
Dumitru Roman
Holger Lausen
Uwe Keller
Co-Authors:
Jos de Bruijn
Christoph Bussler
John Domingue
Dieter Fensel
Martin Hepp
Michael Kifer
Birgitta König-Ries
Jacek Kopecky
Rubén Lara
Eyal Oren
Axel Polleres
James Scicluna
Michael Stollberg

This document is also available in a non-normative PDF version.

Abstract

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

Table of Contents

- 1. Introduction
- 2. Language for Defining WSMO
  - 2.1 The Meta-Model Layers for Defining WSMO
  - 2.2 Identifiers
  - 2.3 Values and Data Types
- 3. WSMO Top-Level Elements
- 4. Ontologies
  - 4.1 Non-Functional Properties
  - 4.2 Importing Ontologies
  - 4.3 Using Mediators
  - 4.4 Concepts
  - 4.5 Relations
  - 4.6 Functions
  - 4.7 Instances
  - 4.8 Axioms
- 5. Web Service Descriptions
  - 5.1 Capability
  - 5.2 Interface
- 6. Goals
- 7. Mediators
- 8. Logical Language for Defining Formal Statements in WSMO
  - 8.1 Variable Identifiers
  - 8.2 Basic Vocabulary and Terms
  - 8.3 Logical Expressions
- 9. Non-Functional Properties
- 10. Conclusions and Further Directions
- References
- Acknowledgements
- Appendix A. Conceptual Model of WSMO
- Appendix B. UML Class Diagrams for WSMO Elements

1. Introduction

This document presents an ontology called Web Service Modeling Ontology (WSMO) for describing various aspects related to Semantic Web Services. Taking the Web Service Modeling Framework (WSMF) [Fensel & Bussler, 2002] as a starting point, we refine and extend this framework, and develop a formal ontology and language. WSMF [Fensel & Bussler, 2002] consists of four different main elements for describing semantic Web Services: (1) ontologies that provide the terminology used by other elements, (2) goals that define the problems that should be solved by Web Services, (3) Web Services descriptions that define various aspects of a Web Service, and (4) mediators which bypass interpretability problems.

WSMO provides ontological specifications for the core elements of Semantic Web services. In fact, Semantic Web services aim at an integrated technology for the next generation of the Web by combining Semantic Web technologies and Web services, thereby turning the Internet from a information repository for human consumption into a world-wide system for distributed Web computing. Therefore, appropriate frameworks for Semantic Web services need to integrate the basic Web design principles, those defined for the Semantic Web, as well as design principles for distributed, service-orientated computing of the Web. WSMO is therefore based on the following design principles:

- **Web Compliance** - WSMO inherits the concept of URI (Universal Resource Identifier) for unique identification of resources as the essential design principle of the Word Wide Web. Moreover, WSMO adopts the concept of Namespaces for denoting consistent information spaces, supports XMLand other W3C Web technology recommendations, as well as the decentralization of resources.
- **Ontology-Based** - Ontologies are used as the data model throughout WSMO, meaning that all resource descriptions as well as all data interchanged during service usage are based on ontologies. Ontologies are a widely accepted state-of-the-art knowledge representation, and have thus been identified as the central enabling technology for the Semantic Web. The extensive usage of ontologies allows semantically enhanced information processing as well as support for interoperability; WSMO also supports the ontology languages defined for the Semantic Web.
- **Strict Decoupling** - Decoupling denotes that WSMO resources are defined in isolation, meaning that each resource is specified independently without regard to possible usage or interactions with other resources. This complies with the open and distributed nature of the Web.
- **Centrality of Mediation** - As a complementary design principle to strict decoupling, mediation addresses the handling of heterogeneities that naturally arise in open environments. Heterogeneity can occur in terms of data, underlying ontology, protocol or process. WSMO recognizes the importance of mediation for the successful deployment of Web services by making mediation a first class component of the framework.
- **Ontological Role Separation** - Users, or more generally clients, exist in specific contexts which will not be the same as for available Web services. For example, a user may wish to book a holiday according to preferences for weather, culture and childcare, whereas Web services will typically cover airline travel and hotel availability. The underlying epistemology of WSMO differentiates between the desires of users or clients and available services.
- **Description versus Implementation** - WSMO differentiates between the descriptions of Semantic Web services elements (description) and executable technologies (implementation). While the former requires a concise and sound description framework based on appropriate formalisms in order to provide a concise for semantic descriptions, the latter is concerned with the support of existing and emerging execution technologies for the Semantic Web and Web services. WSMO aims at providing an appropriate ontological description model, and to be complaint with existing and emerging technologies.
- **Execution Semantics** - In order to verify the WSMO specification, the formal execution semantics of reference implementations like WSMX as well as other WSMO-enabled systems provide the technical realization of WSMO.
- **Service versus Web service** A Web service is a computational entity which is able (by invocation) to achieve a users goal. A service in contrast is the actual value provided by this invocation [Baida et al., 2004], [Preist, 2004] [2]. WSMO provides means to describe Web services that provide access (searching, buying, etc.) to services. WSMO is designed as a means to describe the former and not to replace the functionality of the latter.

This document is organized as follows: Section 2 describes the meta-model structure and language used for defining WSMO based on the Meta-Object Facilities (MOF). We introduce the top level elements of WSMO in Section 3, which are then further refined: Section 4 outlines ontologies, Section 5 discusses Web service descriptions, Section 6 explains WSMO goals, and Section 7 defines mediators. Section 8 defines 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 9 describes the non-functional properties used in the definition of different WSMO elements, and Section 10 presents our conclusions and suggestions for further work.

Finally, the appendix provide an overview of the complete WSMO conceptual model in MOF-style (in Appendix A) as well as in the form of UML Class Diagrams (in Appendix B.)

For a more explanatory document on WSMO we refer to the WSMO Primer [Feier, 2004]. 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] and for the formal representation languages we refer to The WSML Family of Representation Languages [De Bruijn, 2004].

Besides the WSMO working group there are two more working groups involved in the WSMO initiative: The WSML working group focusing on language issues and developing an adequate Web Service Modeling Language with various sublanguages, and the WSMX working group that is concerned with designing and building a reference implementation of an execution environment for WSMO.

Note: The vocabulary defined by WSMO is fully extensible. It is based on URIs with optional fragment identifiers (URI references, or URIrefs) [Berners-Lee et al, 1998]. URI references are used for naming all kinds of things in WSMO. The default namespace of this document is `wsmo:http://www.wsmo.org/2004/d2/`. Furthermore this document uses the following namespace abbreviations: `dc:http://purl.org/dc/elements/1.1#`, `xsd:http://www.w3.org/2001/XMLSchema#`, and `foaf:http://xmlns.com/foaf/0.1/.`

Note: "Web Service" and "Service" are used interchangeably in this document; they are meant to represent systems designed to support interoperable machine-to-machine interactions over the internet.

2. Language for Defining WSMO

In this section we introduce the language used to define WSMO. Besides the Meta-Model Layers we explain the use of identifiers and data type values.

2.1 The Meta-Model Layers for Defining WSMO

WSMO is a meta-model for Semantic Web Services related aspects. The Meta-Object Facility (MOF) [OMG, 2002] specification is used to specify this model. MOF defines an abstract language and framework for specifying, constructing, and managing technology neutral meta-models.

MOF defines a metadata architecture consisting of four layers, namely:

- The *information layer* comprises the data to be described.
- The *model layer* comprises the metadata that describes data in the information layer.
- The *meta-model layer* comprises the descriptions that define the structure and semantics of the metadata.
- The *meta-meta-model layer* comprises the description of the structure and semantics of meta-metadata.

In terms of the four MOF layers,

- the language defining WSMO corresponds to the meta-meta model layer,
- WSMO itself constitutes the meta-model layer,
- the actual ontologies, Web services, goals, and mediators specifications constitute the model layer,
- and the actual data described by the ontologies and exchanged between Web services constitute the information layer.

The most frequently used MOF meta-modeling construct in the definition of WSMO is the **Class** construct (and implicitly its class-generalization **sub-Class** construct), together with its **Attributes**, the **type** of the Attributes and their multiplicity specifications. When defining WSMO, the following assumptions are made:

- Every Attribute has its multiplicity set to multi-valued by default; when an Attribute requires its multiplicity to be set to single-valued, this will be explicitly stated in the listings below.
- For some WSMO elements it is necessary to define attributes taking values from the union of several types, a feature that is not directly supported by MOF meta-modeling constructs; this can be simulated in MOF by defining a new Class as super-Class of all the types required in the definition of the attribute (that represents the union of the single types), with the Constraint that each instance of this new Class is an instance of at least one of the types which are used in the union. This new Class in WSMO is defined by curly brackets, enumerating the Classes representing the required types of the definition of the attributes in between.

For describing the WSMO conceptual model we will use MOF-style listings throughout this document, using the **Class**, **sub-Class Attributes** and **type** keywords. Note that, for better readability we additionally provide a graphical illustration of the the conceptual model of WSMO in the form of UML class diagrams in Appendix B.

2.2 Identifiers

Every WSMO element is identified by one of the following identifiers:

- **URI references**

  WSMO is based on the principle of identifying entities using Web identifiers (called Uniform Resource Identifiers [Berners-Lee et al, 1998]). Everything in WSMO is by default denoted by an URI, except when it classifies itself as Literal, Variable or Anonymous Id. Using URIs does not limit WSMO from making 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. URIs can be expressed as follows: full URIs: e.g. http://www.wsmo.org/2004/d2/ or qualified Names (QNames) that are resolved using namespace declarations. For more details on QNames, we refer to [Bray et al., 1999].

- **Anonymous IDs**

  Anonymous IDs can be numbered (`_#1, _#2, ...`) or unnumbered (`_#`). They represent identifiers. The same numbered Anonymous Id represents the same identifier within the same scope (logical expression), otherwise Anonymous IDs represent different identifiers [Yang & Kifer, 2003]. Anonymous IDs can be used to denote objects that exists, but do not 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 a connecting object between John and "hitchhikersstreet" and "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 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 within implementations.

2.3 Values and Data Types

In WSMO, literals are used to identify values such as numbers by means of a lexical representation. Literals are either plain literals or typed literals. A literal can be typed by a XML data type (e.g. to xsd:integer). Formally, such a data type `d` 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 called data type predicates and are defined in more detail in the WSML Family of Representation Languages [De Bruijn, 2004].

3. WSMO Top-Level Elements

Following the main elements identified in the Web Service Modeling Framework, WSMO identifies four top level elements as the main concepts which have to be described in order to describe Semantic Web services.

Listing 1. WSMO Definition

```
Class wsmoTopLevelElement
      hasNonFunctionalProperties type nonFunctionalProperties

Class ontology sub-Class wsmoTopLevelElement
Class webService sub-Class wsmoTopLevelElement
Class goal sub-Class wsmoTopLevelElement
Class mediator sub-Class wsmoTopLevelElement
```

Ontologies provide the terminology used by other WSMO elements to describe the relevant aspects of the domains of discourse; they are described in detail in Section 4.

Web services describes the computational entity providing access to services that provide some value in a domain. These descriptions comprise the capabilities, interfaces and internal working of the Web service (as further described in Section 5). All these aspects of a Web Service are described using the terminology defined by the ontologies.

Goals represent user desires, for which fulfilment could be sought by executing a Web service. Ontologies can be used for the domain terminology to describe the relevant aspects. Goals model the user view in the Web service usage process and are therefore a separate top-level entity in WSMO described in detail in Section 6.

Finally, **Mediators** describe elements that overcome interoperability problems between different WSMO elements. Mediators are the core concept to resolve incompatibilities on the data, process and protocol level, i.e. in order to resolve mismatches between different used terminologies (data level), in how to communicate between Web services (protocol level) and on the level of combining Web services (and goals) (process level). These are described in detail in Section 7.

4. Ontologies

In WSMO, Ontologies are the key to linking 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 an agreed common terminology by providing concepts, and relationships beween the concepts. In order to capture semantic properties of relations and concepts, an ontology generally also provides a set of axioms, which are expressions in some logical language. The following listing consists of the definition of the WSMO ontology element and the following sub-sections describe in more detail the attributes of the WSMO ontology element.

Listing 2. Ontology Definition

Class ontology
    hasNonFunctionalProperties **type** nonFunctionalProperties
    importsOntology **type** ontology
    usesMediator **type** ooMediator
    hasConcept **type** concept
    hasRelation **type** relation
    hasFunction **type** function
    hasInstance **type** instance
    hasAxiom **type** axiom

4.1 Non-Functional Properties

The following non-functional properties recommended for characterizing ontologies are: Contributor, Coverage, Creator, Date, Description, Format, Identifier, Language, Owner, Publisher, Relation, Rights, Source, Subject, Title, Type, Version.

4.2 Importing Ontologies

Building an ontology for some particular problem domain can be a rather cumbersome and complex task. One standard way of dealing with the complexity is modularization. Importing Ontologies allows a modular approach for ontology design. Importing can be used as long as no conflicts need to be resolved, otherwise an `ooMediator` needs to be used.

4.3 Using 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 in more detail in Section 7.

4.4 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 be a subconcept of several (possibly none) direct superconcepts as specified by the isA-relation.

Listing 3. Concept Definition

Class concept
    hasNonFunctionalProperties **type** nonFunctionalProperties
    hasSuperConcept **type** concept
    hasAttribute **type** attribute
    hasDefinition **type** logicalExpression *`multiplicity = single-valued`*

Non-Functional Properties
: The non-functional properties recommended are: Contributor, Coverage, Creator, Date, Description, Identifier, Relation, Source, Subject, Title, Type, Version.

Superconcept
: There is a finite number of concepts that serve as a superconcept for some concept. Being a sub-concept 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 each of its superconcepts.

Attribute

Each concept provides a (possibly empty) set of attributes that represent named slots for data values for instances. They can 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 4. Attribute Definition

```
Class attribute
  hasNonFunctionalProperties type nonFunctionalProperties
  hasRange type concept multiplicity = single-valued
```

Non-Functional Properties

The non-functional properties recommended are: Contributor, Coverage, Creator, Date, Description, Identifier, Relation, Source, Subject, Title, Type, Version.

Range

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

Definition

The definition is a logical expression (see Section 8) which can be used to define formally the semantics of the concept. More precisely, the logical expression defines (or restricts, respectively) 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 implies l-expr(?x) )`
- `forAll ?x ( ?x memberOf C impliedBy l-expr(?x) )`
- `forAll ?x ( ?x memberOf C equivalent l-expr(?x) )`

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

The first example expresses that there is a necessary condition for membership in the extension of the concept. The second example expresses that there is a sufficient condition, and the third case means that there is a sufficient and necessary condition for an object being an element of the extension of the concept.

4.5 Relations

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

Listing 5. Relation Definition

```
Class relation
  hasNonFunctionalProperties type nonFunctionalProperties
  hasSuperRelation type relation
  hasParameter type parameter
  hasDefinition type logicalExpression multiplicity = single-valued
```

Non-Functional Properties

The non-functional properties recommended are: Contributor, Coverage, Creator, Date, Description, Identifier, Relation, Source, Subject, Title, Type, Version.

Superrelation

A finite set of relations of which the defined relation is declared as being a subrelation. Being a subrelation of some other relation in particular means that the relation inherits the signature of this superrelation and the corresponding constraints. Furthermore, the set of tuples belonging to the relation (the extension of the relation, respectively) is a subset of each of the extensions of the superrelations.

Parameter

The list of parameters.

Listing 6. Parameter Definition

```
Class parameter
  hasNonFunctionalProperties type nonFunctionalProperties
  hasDomain type concept multiplicity = single-valued
```

Non-Functional Properties

The non-functional properties recommended are: Contributor, Coverage, Creator, Date, Description, Identifier, Relation, Source, Subject, Title, Type, Version.

Domain

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

Definition

A logical expression (see Section 8) defining the set of instances (n-ary tuples, if n is the arity of the relation) of the relation. If the parameters are specified, the relation is represented by an n-ary predicate symbol with named arguments (see Section 8) (where n is the number of parameters of the relation) and the identifier of the relation is used as the name of the predicate symbol. If R is the identifier denoting the relation and the parameters are specified, then the logical expression takes one of the following forms:

- `forAll ?v1,...,?vn ( R[p1 hasValue ?v1,...,pn hasValue ?vn] implies l-expr(?v1,...,?vn) )`
- `forAll ?v1,...,?vn ( R[p1 hasValue ?v1,...,pn hasValue ?vn] impliedBy l-expr(?v1,...,?vn) )`
- `forAll ?v1,...,?vn ( R[p1 hasValue ?v1,...,pn hasValue ?vn] equivalent l-expr(?v1,...,?vn) )`

If the parameters are not specified, then the relation is represented by a predicate symbol (see Section 8) where the identifier of the relation is used as the name of the predicate symbol. If R is the identifier denoting the relation and the parameters are not specified, then the logical expression takes one of the following forms:

- `forAll ?v1,...,?vn ( R(?v1,...,?vn) implies l-expr(?v1,...,?vn) )`
- `forAll ?v1,...,?vn ( R(?v1,...,?vn) impliedBy l-expr(?v1,...,?vn) )`
- `forAll ?v1,...,?vn ( R(?v1,...,?vn) equivalent l-expr(?v1,...,?vn) )`

`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.

Using `implies` means there is a necessary condition for instances `?v1,...,?vn` to be related. Using `impliedBy` means that there is a sufficient condition and using `equivalent` means that there is a sufficient and necessary condition for instances `?v1,...,?vn` being related.

4.6 Functions

A function is a special relation, with an unary range and a n-ary domain (parameters inherited from the relation), where the range value is functionally dependent on the domain values. In particular, the following constraint must hold (where F is the name of the function and != stands for inequality):

```
forAll ?x1,...,?xn,?r1,?r2 (false impliedBy F(?x1,...,?xn,?r1) and F(?x1,...,?xn,?r2) and ?r1 != ?r2)
```

In contrast to a function symbol, a function is not only a syntactical entity but has a defined semantics that allows to actually evaluate the function if concrete input values for the parameters are given. That means that we can actually substitute the (ground) function term in some expression by its concrete value. Functions for example can be used to represent and exploit built-in predicates of common datatypes. Their semantics can be captured externally by means of an oracle, or can be formalized by assigning a logical expression to the `hasDefinition` property inherited from relation.

Listing 7. Function Definition

```
Class function sub-Class relation
    hasRange type concept multiplicity = single-valued
```

Range

A concept constraining the possible return values of the function.

The logical representation of a function is almost the same as that of relations, whereby the result value of a function (respectively the value of a function term) has to be represented explicitly: the function is represented by a (n+1)-ary predicate symbol with named arguments (see Section 8) (where n is the number of arguments of the function) and the identifier of the function is used as the name of the predicate. In particular, the names of the parameters of the corresponding relation symbol are the names of the parameters of the function as well as one additional parameter range for denoting the value of the function term with the given parameter values. If the parameters are not specified, the function is represented by a predicate symbol with ordered arguments, and by convention the first argument specifies the value of the function term with given argument values.

If `F` is the identifier denoting the function and `p1,..., pn` is the set of parameters of the function then the logical expression for defining the semantics of the function (inherited from relation) can for example take the form

```
forAll ?v1,...,?vn,?range ( F[p1 hasValue ?v1,...,pn hasValue ?vn, range hasValue ?range]
                            equivalent l-expr(?v1,...,?vn,?range) )
```

where `l-expr(?v1,...,?vn,?range)` is a logical expression with precisely `?v1,...,?vn,?range` as its free variables and `p1,...,pn` are the names of the parameters of the function. Clearly, in order to prevent ambigiuities, `range` may not be used as the name for a parameter of a function in order to prevent ambiguities.

4.7 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 8. Instance Definition

Class instance
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasType **type** concept
  hasAttributeValues **type** attributeValue

Non-Functional Properties
The non-functional properties recommended are: Contributor, Coverage, Creator, Date, Description, Identifier, Relation, Source, Subject, Title, Type, Version.

Type
The concept of which this instance is an instance.

Attribute Value
The attribute values for the single attributes defined in the concept. For each attribute of the concept, this instance is assigned so there can be one or more corresponding attribute values. These values have to be compatible with the corresponding type declaration in the concept definition.

Listing 9. Attribute Value Definition

Class attributeValue
  hasAttribute **type** attribute *`multiplicity = single-valued`*
  hasValue **type** {instance, literal, anonymousId}

Attribute
The attribute this value refers to.

Value
An instance, 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 10. Relation Instance Definition

Class relationInstance
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasType **type** relation
  hasParameterValue **type** parameterValue

Non-Functional Properties
The non-functional properties recommended are: Contributor, Coverage, Creator, Date, Description, Identifier, Relation, Source, Subject, Title, Type, Version.

Type
The relation this instance belongs to.

Parameter Value
A set of parameter values 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 that are specified in the corresponding relation.

Listing 11. Parameter Value Definition

Class parameterValue
  hasParameter **type** parameter *`multiplicity = single-valued`*
  hasValue **type** {instance, literal, anonymousId} *`multiplicity = single-valued`*

Parameter
The parameter this value refers to.
Value
An instance, literal or anonymous ID representing the actual value of an instance for a specific paramter.

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

4.8 Axioms

An axiom is a logical expression together with its non-functional properties.

Listing 12. Axiom Definition

```
Class axiom
    hasNonFunctionalProperties type nonFunctionalProperties
    hasDefinition type logicalExpression
```

Non-Functional Properties
The non-functional properties recommended are: Contributor, Coverage, Creator, Date, Description, Identifier, Relation, Source, Subject, Title, Type, Version.
Definition
The actual statement captured by the axiom is defined by a formula in a logical language as described in Section 8.

5. Web Service Descriptions

WSMO Web service descriptions consist of functional, non-functional and the behavioral aspects of a Web service. A Web service is a computational entity which is able (by invocation) to achieve a users goal. A service in contrast is the actual value provided by this invocation. Thereby a Web service might provide different services, such as for example Amazon can be used for acquiring books as well as to find out an ISBN number of a book.

In the following, we describe the elements of a Web service description:

Listing 13. Web Service Description Definition

Class webService
    hasNonFunctionalProperties **type** nonFunctionalProperties
    importsOntology **type** ontology
    usesMediator **type** {ooMediator, wwMediator}
    hasCapability **type** capability *multiplicity = single-valued*
    hasInterface **type** interface

Non-Functional Properties
The non-functional properties recommended are: Accuracy, Contributor, Coverage, Creator, Date, Description, Financial, Format, Identifier, Language, Network-related QoS, Owner, Performance, Publisher, Relation, Reliability, Rights, Robustness, Scalability, Security, Source, Subject, Title, Transactional, Trust, Type, Version.

Importing Ontology
Used to import ontologies as long as no conflicts need to be resolved.

Using Mediator
A Web service can import ontologies using ontology mediators (`ooMediator`) when steps for aligning, merging, and transforming imported ontologies are needed. A Web service can use wwMediators to deal with process and protocol mediation.

5.1 Capability

A capability defines the Web service by means of its functionality.

Listing 14. Capability Definition

Class capability
    hasNonFunctionalProperties **type** nonFunctionalProperties
    importsOntology **type** ontology
    usesMediator **type** {ooMediator, wgMediator}
    hasSharedVariables **type** sharedVariables
    hasPrecondition **type** axiom
    hasAssumption **type** axiom
    hasPostcondition **type** axiom
    hasEffect **type** axiom

Non-Functional Properties
The non-functional properties recommended are: Accuracy, Contributor, Coverage, Creator, Date, Description, Financial, Format, Identifier, Language, Network-related QoS, Owner, Performance, Publisher, Relation, Reliability, Rights, Robustness, Scalability, Security, Source, Subject, Title, Transactional, Trust, Type, Version.

Importing Ontology
Used to import ontologies as long as no conflicts need to be resolved.

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

Shared Variables
Shared Variables represent the variables that are shared between preconditions, postconditons, assumptions and effects. They are all quantified variables in the formula that concatenates assumptions, preconditions, postconditions, and effects.

If `?v1,...,?vn` are the shared variables defined in a capability, and `pre(?v1,...,?vn)`, `ass(?v1,...,?vn)`, `post(?v1,...,?vn)` and `eff(?v1,...,?vn)`, are used to denote the formulae defined by the preconditions, assumptions, postconditions, and effects respectively, then the following holds:

```
forAll ?v1,...,?vn ( pre(?v1,...,?vn) and ass(?v1,...,?vn)
   implies post(?v1,...,?vn) and eff(?v1,...,?vn)).
```

Precondition
Preconditions specify the information space of the Web service before its execution.
Assumption
Assumptions describe the state of the world before the execution of the Web service.
Postcondition
Postconditions describe the information space of the Web service after the execution of the Web service.
Effect
Effects describe the state of the world after the execution of the Web service.

5.2 Interface

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

- choreography decomposes a capability in terms of interaction with the Web service.
- orchestration decomposes a capability in terms of functionality required from other Web services.

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 Web service providers [1].

An interface is defined by the following properties:

Listing 15. Interface Definition

```
Class interface
    hasNonFunctionalProperties type nonFunctionalProperties
    importsOntology type ontology
    usesMediator type ooMediator
    hasChoreography type choreography
    hasOrchestration type orchestration
```

Non-Functional Propreties
The non-functional properties recommended are: Accuracy, Contributor, Coverage, Creator, Date, Description, Financial, Format, Identifier, Language, Network-related QoS, Owner, Performance, Publisher, Relation, Reliability, Rights, Robustness, Scalability, Security, Source, Subject, Title, Transactional, Trust, Type, Version.
Importing Ontology
Used to import ontologies as long as no conflicts need to be resolved.
Using Mediator
An interface can import ontologies using ontology mediators (`ooMediator`) when steps for aligning, merging, and transforming imported ontologies are needed.
Choreography
Choreography provides the necessary information to enable communication with the service from the client point of view [Roman et al., 2005].
Orchestration
Orchestration describes how the service makes use of other services in order to achieve its capability [Roman et al., 2005] .

6. Goals

Goals are representations of an objective for which fulfillment is sought through the execution of a Web service. Goals can be descriptions of Web services that would potentially satisfy the user desires. The following listing presents the goal definition:

Listing 16. Goal Definition

```
Class goal
    hasNonFunctionalProperties type nonFunctionalProperties
    importsOntology type ontology
    usesMediator type {ooMediator, ggMediator}
    requestsCapability type capability multiplicity = single-valued
    requestsInterface type interface
```

Non-Functional Properties
: The non-functional properties recommended are: Accuracy, Contributor, Coverage, Creator, Date, Description, Financial, Format, Identifier, Language, Network-related QoS, Owner, Performance, Publisher, Relation, Reliability, Rights, Robustness, Scalability, Security, Source, Subject, Title, Transactional, Trust, Type, Type of Match, Version.

Importing Ontology
: Used to import ontologies as long as no conflicts need to be resolved.

Using Mediator
: A goal can import ontologies by using ontology mediators (`ooMediator`) in case assistance 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 (`ggMediator`). For a detailed account on mediators we refer to Section 7.

Capability
: The capability of the Web services the user would like to have.

Interface
: The interface of the Web service the user would like to have and interact with.

7. 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 or state equivalence if both goals are substitutable.
- ooMediators: mediators that import ontologies and resolve possible representation mismatches between ontologies.
- wgMediators: mediators that link Web services to goals, meaning that the Web service (totally or partially) fulfills the goal to which it is linked. wgMediators may explicitly state the difference between the two entities and map different vocabularies (through the use of ooMediators).
- wwMediators: mediators linking two Web services.

The mediators are defined as follows:

Listing 17. Mediators Definition

```
Class mediator
    hasNonFunctionalProperties type nonFunctionalProperties
    importsOntology type ontology
    hasSource type {ontology, goal, webService, mediator}
    hasTarget type {ontology, goal, webService, mediator}
    hasMediationService type {goal, webService, wwMediator}

Class ooMediator sub-Class mediator
    hasSource type {ontology, ooMediator}

Class ggMediator sub-Class mediator
    usesMediator type ooMediator
    hasSource type {goal, ggMediator}
    hasTarget type {goal, ggMediator}

Class wgMediator sub-Class mediator
    usesMediator type ooMediator
    hasSource type {webService, goal, wgMediator, ggMediator}
    hasTarget type {webService, goal, ggMediator, wgMediator}

Class wwMediator sub-Class mediator
    usesMediator type ooMediator
    hasSource type {webService, wwMediator}
    hasTarget type {webService, wwMediator}
```

Non-Functional Properties
: The non-functional properties recommended are: Accuracy, Contributor, Coverage, Creator, Date, Description, Financial, Format, Identifier, Language, Network-related QoS, Owner, Performance, Publisher, Relation, Reliability, Rights, Robustness, Scalability, Security, Source, Subject, Title, Transactional, Trust, Type, Version.

Importing Ontology
: Used to import ontologies as long as no conflicts need 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 target of the mediator.

Mediation Service
: The mediation service 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.

Using Mediator
: 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 Web service capabilities and align different heterogeneous ontologies.

A further remark on wgMediators is in place: A wgMediator can - depending on which is the source and target

of the mediator - have two different functions: (1) A goal is linked to a Web service via its choreography interface meaning that the Web service (totally or partially) fulfills the goal to which it is linked (2) A Web service links to a goal via its orchestration interface meaning that the Web service needs this goal to be resolved in order to fulfil the functionality described in its capability.

Notice that there are two principal ways of relating mediators with other entities in the WSMO model: (1) an entity can specify a relation with a mediator through the has `usesMediator` 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 Web services between these particular entities. WSMO does not prescribe the type of use of mediators and therefore provides maximum flexibility in the use of mediators and thus allows for loose coupling between Web services, goals, and ontologies.

8. 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 logical expressions. The semantics of this language is defined formally by the WSML working group in a separate document.

Section 8.1 introduces the identifiers recommended for variables in WSMO. Section 8.2 gives the definition of the basic vocabulary and the set of terms for building logical expression. Section 8.3 defines the most basic formulas (i.e. atomic formulae) which allows us to eventually define the set of logical expressions.

8.1 Variable Identifiers

Apart from the identifiers (URIs and anonymous) defined in Section 2.1 and values defined in Section 2.2, logical expressions in WSMO can also identify variables. Variable names are strings that start with a question mark '`?`', followed by any positive number of symbols in {`a-z`, `A-Z`, `0-9`, `_`, `-`}, for example `?var` or `?lastValue_Of`.

8.2 Basic Vocabulary and Terms

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

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

- A (possibly infinite) set of **Uniform Resource Identifiers** *URI*.
- A (possibly infinite) set of **anonymous IDs** *AnID.*
- A (possibly infinite) set of **literals** *Lit*.
- A (possibly infinite) set of **variables** *Var*.
- A (possibly infinite) set of **function symbols** (object constructors, respectively) *FSym* which is a subset of *URI.*
- A (possibly infinite) set of **predicate symbols** *PSym* which is a subset of *URI.*
- A (possibly 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`, `implies`,`impliedBy`, `equivalent` , `forAll`, `exists`.
- All these sets are assumed to be *mutually distinct* (as long as no subset relationship has been explicitly stated).
- For each symbol *S* in *FSym*, *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 *PSymNamed*, we assume that there is a corresponding **set of parameter names** *parNames(S)* defined, which gives the names of the 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 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 `t1, ..., tn` are terms, then `f(t1, ..., tn)` 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.

8.3 Logical Expressions

We extend the previous definition (Definition 2) to the **set of (complex) logical expressions** (or formulae, respectively) **L(V)** (over vocabulary V) as follows:

Definition 3. A **simple logical expression** in *L(V)* (or atomic formula) is inductively defined as follows:

- If `p` is a predicate symbol in *PSym* with arity(p) = n and `t1, ..., tn` are terms, then `p(t1, ..., tn)` 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) = {`p1, ...,pn`} and `t1, ...,tn` are terms, then `R[p1 hasValue t1, ..., pn hasValue tn]` 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 ofType T]` is a simple logical expression in *L(V)* .
- If `P`, `ATT, T1`,...,`Tn` (where n >= 1) are terms in *Term(V)*, then `P[ATT ofTypeSet (T1,...,Tn)]` 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 `C1`, `C2` are terms in *Term(V)*, then `C1 subConceptOf C2` is a simple logical expression in *L(V)*.
- If `R1`, `C2` are predicate symbols in *PSym* or *PSymNamed* with the same signature, then `R1 subRelationOf R2` is a simple logical expression on *L(V)*.
- If `O`, `V`, `ATT` are terms in *Term(V)*, then `O[ATT hasValue V]` is a simple logical expression in *L(V)*.
- If `O`, `V1`,...,`Vn`, `ATT` (where n >=1) are terms in *Term(V)*, then `O[ATT hasValues {V1,...,Vn}]` is a simple logical expression in *L(V)*.
- If `T1` and `T2` are terms in *Term(V)*, then `T1 = T2` 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. The arguments are explicitely defined by the associated parameter names. Obviously, this has consequences for unification algorithms.
- true and false denote atomic statements which are always true (or false, respectively)
- `C[ATT ofType 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 ofTypeSet (`T1,...Tn)]. It defines a constraint on the possible values that instances of class `C` may take for property `ATT` to values of types T1, .., Tn. 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`.
- `C1` subConceptOf `C2` is true iff concept `C1` is a subconcept of concept `C2`, that means the extension of concept `C1` is a subset of the extension of concept `C2`.
- `O[ATT` hasValue `V]` is true if the element denoted by `O` takes value `V` under property `ATT`.
- Similar for the simple logical expression `O[ATT hasValues {V1,...,Vn}]`: The expression holds if the set of values that the element `O` takes for property `ATT` includes all the values `V1,...Vn`. That means the set of values of `O` for property `ATT` is a superset of the set `{V1, ..., Vn}`.
- `T1 = T2` is true, if both terms `T1` and `T1` denote the same element of the universe.

Definition 4. Definition 3 is extended to **complex logical expressions** 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 `L1` and `L2` are logical expressions in *L(V)* and `op` is one of the logical connectives in { `or,and,implies, impliedBy, equivalent` }, then `L1 op L2` 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, respectively) 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, impliedBy` 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 `op1` **<** `op2` means that `op2`

binds stronger than `op1`: `implies`, `equivalent`,`impliedBy`**<** `or`**,** `and` **<** `not`.

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

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

A logical expression of the form `false impliedBy L` (commonly used in Logic Programming systems for defining integrity constraints) can be written using the following syntactical shortcut: `constraint L.`

We allow the following syntactic composition of atomic formulas as a syntactic abbreviation for two separate atomic formulas: `C1 subConceptOf C2` and `C1[ATT op V]` can be syntactically combined to `C1[ATT op V] subConceptOf C2`. Additionally, for the sake of backwards compatibility with F-Logic, we also allow the following notation for the combination of the two atomic formulae: `C1 subConceptOf C2 [ATT op V]`. Both abbreviations stand for the set of the two single atomic formulae. The first abbreviation is considered to be the standard abbreviation for combining these two kinds of atomics formulae.

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 predicates with named arguments, instances as terms and axioms as logical expressions.

9. Non-Functional Properties

Non functional properties are used in the definition of WSMO elements. Which non-functional properties apply to which WSMO element is specified in the description of each WSMO element. We recommend most elements of [Weibel et al., 1998].

Non-functional properties are defined in the following way:

Listing 18. Non-Functional Pproperties Definition

```
Class nonFunctionalProperties
    hasAccuracy type accuracy
    hasContributor type dc:contributor
    hasCoverage type dc:coverage
    hasCreator type dc:creator
    hasDate type dc:date
    hasDescription type dc:description
    hasFinancial type financial
    hasFormat type dc:format
    hasIdentifier type dc:identifier
    hasLanguage type dc:language
    hasNetworkRelatedQoS type networkRelatedQoS
    hasOwner type owner
    hasPerformance type performance
    hasPublisher type dc:publisher
    hasRelation type dc:relation
    hasReliability type reliability
    hasRights type dc:rights
    hasRobustness type robustness
    hasScalability type scalability
    hasSecurity type security
    hasSource type dc:source
    hasSubject type dc:subject
    hasTitle type dc:title
    hasTransactional type transactional
    hasTrust type trust
    hasType type dc:type
    hasTypeOfMatch type typeOfMatch
    hasVersion type version
```

Accuracy
Represents the error rate generated by the Web service. It can be measured by the numbers of errors generated in a certain time interval.

Contributor
An entity responsible for making contributions to the content of the element. Examples of `dc:contributor` include a person, an organization, or a Web 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].

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.

Creator
An entity primarily responsible for creating the content of the element. Examples of `dc:creator` include a person, an organization, or a Web 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].

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 using 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 XML Schema Definition (YYYY-MM-DD) [Biron & Malhotra, 2001] and thus one is automatically compliant with XML Schema, too.

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.

Financial

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.

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 using 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 using 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.

Language

A language of the intellectual content of the element.
WSMO Recommendation: We recommend using 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.

Network-Related QoS

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

Owner

A person or organization to which a particular WSMO element belongs.

Performance

Represents how fast a Web 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 Web 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.

Publisher

An entity responsible for making the element available. Examples of `dc:publisher` include a person, an organization, or a Web 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].

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 using 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.

Reliability

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

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 Web 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.

Robustness

Represents the ability of the Web 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 Web service still

function correctly.

Scalability
Represents the ability of the Web 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
Represents the ability of a Web service to provide authentication (entities - users or other Web services - who can access Web service and data should be authenticated), authorization (entities should be authorized so that they only can access the protected Web 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 Web service when a request was serviced), data encryption (data should be encrypted), and non-repudiation (an entity cannot deny requesting a Web service or data after the fact).

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 using URIs as Identifier where possible.

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.

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

Transactional
It represents the transactional properties of the Web service.

Trust
It represents the trust worthiness of a Web service or an ontology.

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 using 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.

Type of Match
The Type of Match desired for a particular goal [Lara, 2004]. Under the Assumption of a set based modelling this can be an exact match, a match where the goal description is a subset of the Web Service description or a match where the Web service description is a subset of the goal description.

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 using the revision numbers of a version control system. Such a system could 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 this "$Revision: 1.113 $".

10. Conclusions and Further Directions

This document presented the Web Service Modeling Ontology (WSMO) for describing several aspects related to services on the Web, 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, and future versions of this document will contain refinements of the mediators.

References

[Baida et al., 2004] Z. Baida, J. Gordijn, B. Omelayenko, and H. Akkermans: *A shared service terminology for online service provisioning*, ICEC '04: Proceedings of the 6th international conference on Electronic commerce, p.1-10, 2004.

[Feier, 2004] C. Feier (Ed.): *WSMO Primer*, WSMO Deliverable D3.1, DERI Working Draft, 2004, latest version available at http://www.wsmo.org/2004/d3/d3.1/.

[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, avaliable 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/.

[de Bruijn., 2004] J. de Bruijn (Ed.): *The WSML Family of Representation Languages*, WSMO Deliverable D16, DERI Working Draft, 2004, latest version available at http://www.wsmo.org/2004/d16/.

[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.

[Gurevich, 1995] Yuri Gurevich: *Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods*, Oxford University Press, 1995, 9--36

[Hayes, 2004] P. Hayes (Ed.): *RDF Semantics*, W3C Recommendation 10 February 2004, 2004.

[IANA, 2002] Intenet 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.

[Keller et. al., 2004] U. Keller, R. Lara, and A. Polleres: *WSMO Web Service Discovery*, WSMO Deliverable D5.1, November 2004, Available from http://www.wsmo.org/2004/d5/d5.1/v0.1/20041112/

[Preist, 2004]Chris Preist: *A conceptual architecture for semantic web services.* In Proceedings of the International Semantic Web Conference 2004 (ISWC 2004), November 2004.

[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, Available from http://dip.semanticweb.org/deliverables/D22ORDIv1.0.pdf

**[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 ParallelDatabases, 12:117-133, 2002.

[OMG, 2002] The Object Management Group: Meta-Object Facility, version 1.4, 2002. Available at http://www.omg.org/technology/documents/formal/mof.htm.

[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.)

[Roman et al., 2005] D. Roman, J. Scicluna and C. Feier: *Ontology-based Choreography and Orchestration of WSMO Services*, WSMO Deliverable D14, DERI Working Draft, 2005, latest version available at http://www.wsmo.org/TR/d14/

[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.

[W3C Glossary, 2004] Hugo Haas, and Allen Brown (editors): Web Services Glossary, W3C Working Group Note 11 February 2004, available at http://www.w3.org/TR/ws-gloss/

Acknowledgments

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

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

Appendix A. Conceptual Elements of WSMO

Class wsmoTopLevelElement
  hasNonFunctionalProperties **type** nonFunctionalProperties

Class ontology **sub-Class** wsmoTopLevelElement
  importsOntology **type** ontology
  usesMediator **type** ooMediator
  hasConcept **type** concept
  hasRelation **type** relation
  hasFunction **type** function
  hasInstance **type** instance
  hasAxiom **type** axiom

Class concept
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasSuperConcept **type** concept
  hasAttribute **type** attribute
  hasDefinition **type** logicalExpression *multiplicity* = *single-valued*

Class attribute
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasRange **type** concept *multiplicity* = *single-valued*

Class relation
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasSuperRelation **type** relation
  hasParameter **type** parameter
  hasDefinition **type** logicalExpression *multiplicity* = *single-valued*

Class parameter
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasDomain **type** concept *multiplicity* = *single-valued*

Class function sub-**Class** relation
  hasRange **type** concept *multiplicity* = *single-valued*

Class instance
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasType **type** concept
  hasAttributeValues **type** attributeValue

Class attributeValue
  hasAttribute **type** attribute *multiplicity* = *single-valued*
  hasValue **type** {instance, literal, anonymousId}

Class relationInstance
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasType **type** relation
  hasParameterValues **type** parameterValue

Class parameterValue
  hasParameter **type** parameter *multiplicity* = *single-valued*
  hasValue **type** {instance, literal, anonymousId} *multiplicity* = *single-valued*

Class axiom
  hasNonFunctionalProperties **type** nonFunctionalProperties
  hasDefinition **type** logicalExpression

Class webService **sub-Class** wsmoTopLevelElement
  importsOntology **type** ontology
  usesMediator **type** {ooMediator, wwMediator}
  hasCapability **type** capability *multiplicity* = *single-valued*
  hasInterface **type** interface

Class capability
hasNonFunctionalProperties **type** nonFunctionalProperties
importsOntology **type** ontology
usesMediator **type** {ooMediator, wgMediator}
hasSharedVariables **type** sharedVariables
hasPrecondition **type** axiom
hasAssumption **type** axiom
hasPostcondition **type** axiom
hasEffect **type** axiom

Class interface
hasNonFunctionalProperties **type** nonFunctionalProperties
importsOntology **type** ontology
usesMediator **type** ooMediator
hasChoreography **type** choreography
hasOrchestration **type** orchestration

Class goal **sub-Class** wsmoTopLevelElement
importsOntology **type** ontology
usesMediator **type** {ooMediator, ggMediator}
requestsCapability **type** capability *multiplicity* = *single-valued*
requestsInterface **type** interface

Class mediator **sub-Class** wsmoTopLevelElement
importsOntology **type** ontology
hasSource **type** {ontology, goal, webService, mediator}
hasTarget **type** {ontology, goal, webService, mediator}
hasMediationService **type** {goal, webService, wwMediator}

Class ooMediator **sub-Class** mediator
hasSource **type** {ontology, ooMediator}

Class ggMediator **sub-Class** mediator
usesMediator **type** ooMediator
hasSource **type** {goal, ggMediator}
hasTarget **type** {goal, ggMediator}

Class wgMediator **sub-Class** mediator
usesMediator **type** ooMediator
hasSource **type** {webService, goal, wgMediator, ggMediator}
hasTarget **type** {webService, goal, ggMediator, wgMediator}

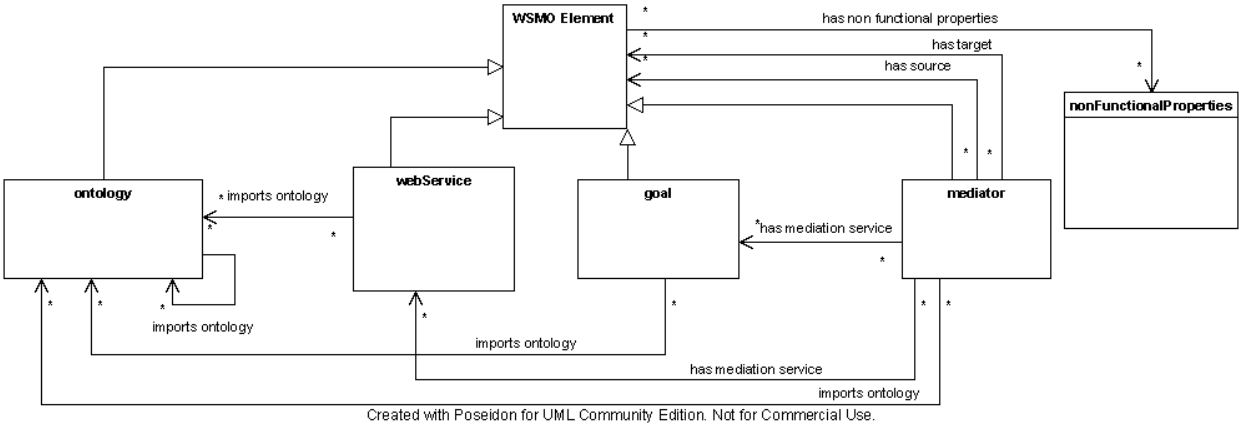
Class wwMediator **sub-Class** mediator
usesMediator **type** ooMediator
hasSource **type** {webService, wwMediator}
hasTarget **type** {webService, wwMediator}

Class nonFunctionalProperties
hasAccuracy **type** accuracy
hasContributor **type** dc:contributor
hasCoverage **type** dc:coverage
hasCreator **type** dc:creator
hasDate **type** dc:date
hasDescription **type** dc:description
hasFinancial **type** financial
hasFormat **type** dc:format
hasIdentifier **type** dc:identifier
hasLanguage **type** dc:language
hasNetworkRelatedQoS **type** networkRelatedQoS
hasOwner **type** owner
hasPerformance **type** performance
hasPublisher **type** dc:publisher
hasRelation **type** dc:relation
hasReliability **type** reliability

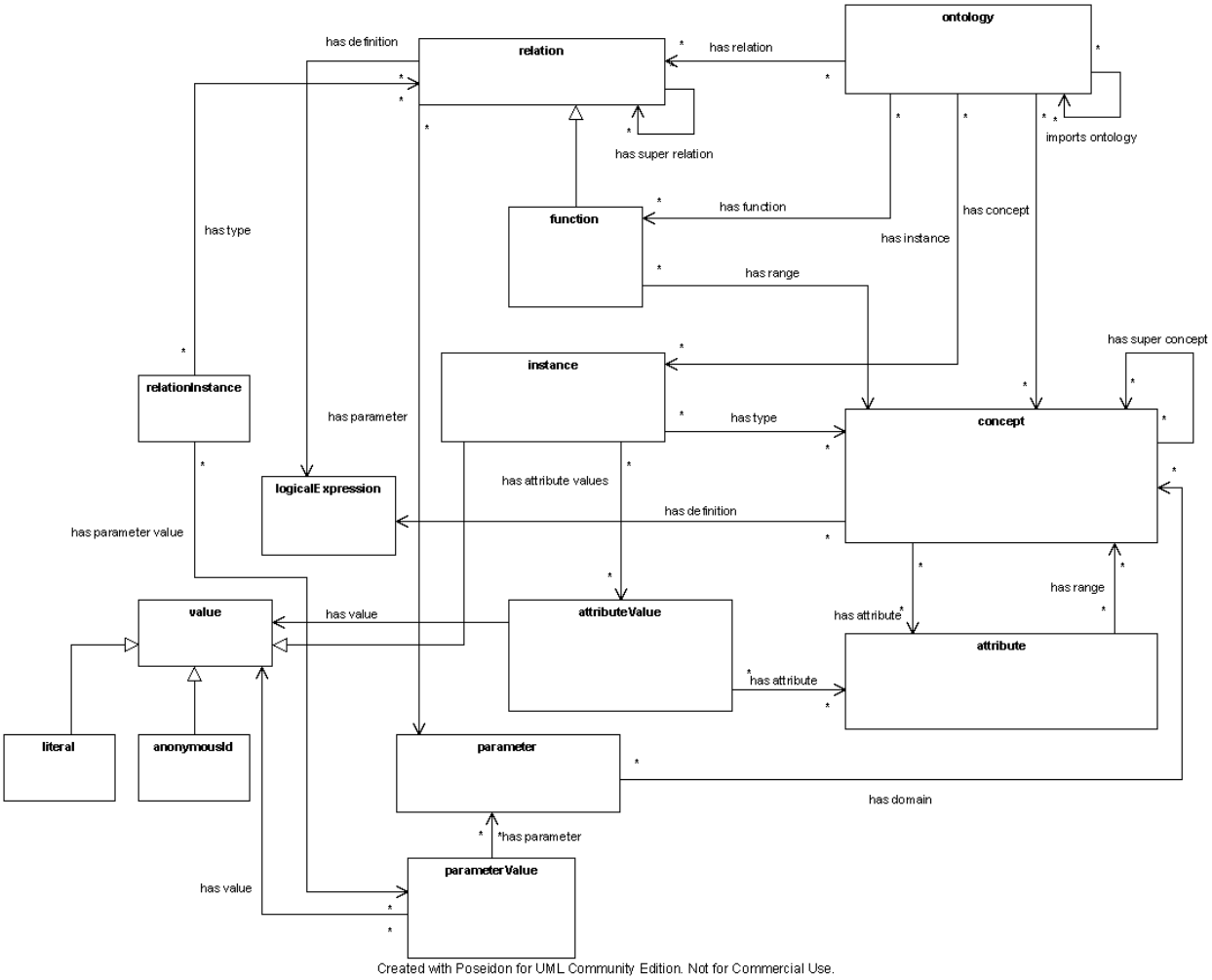
hasRights **type** dc:rights
hasRobustness **type** robustness
hasScalability **type** scalability
hasSecurity **type** security
hasSource **type** dc:source
hasSubject **type** dc:subject
hasTitle **type** dc:title
hasTransactional **type** transactional
hasTrust **type** trust
hasType **type** dc:type
hasTypeOfMatch **type** typeOfMatch
hasVersion **type** version

Appendix B. UML Class Diagrams for WSMO Elements

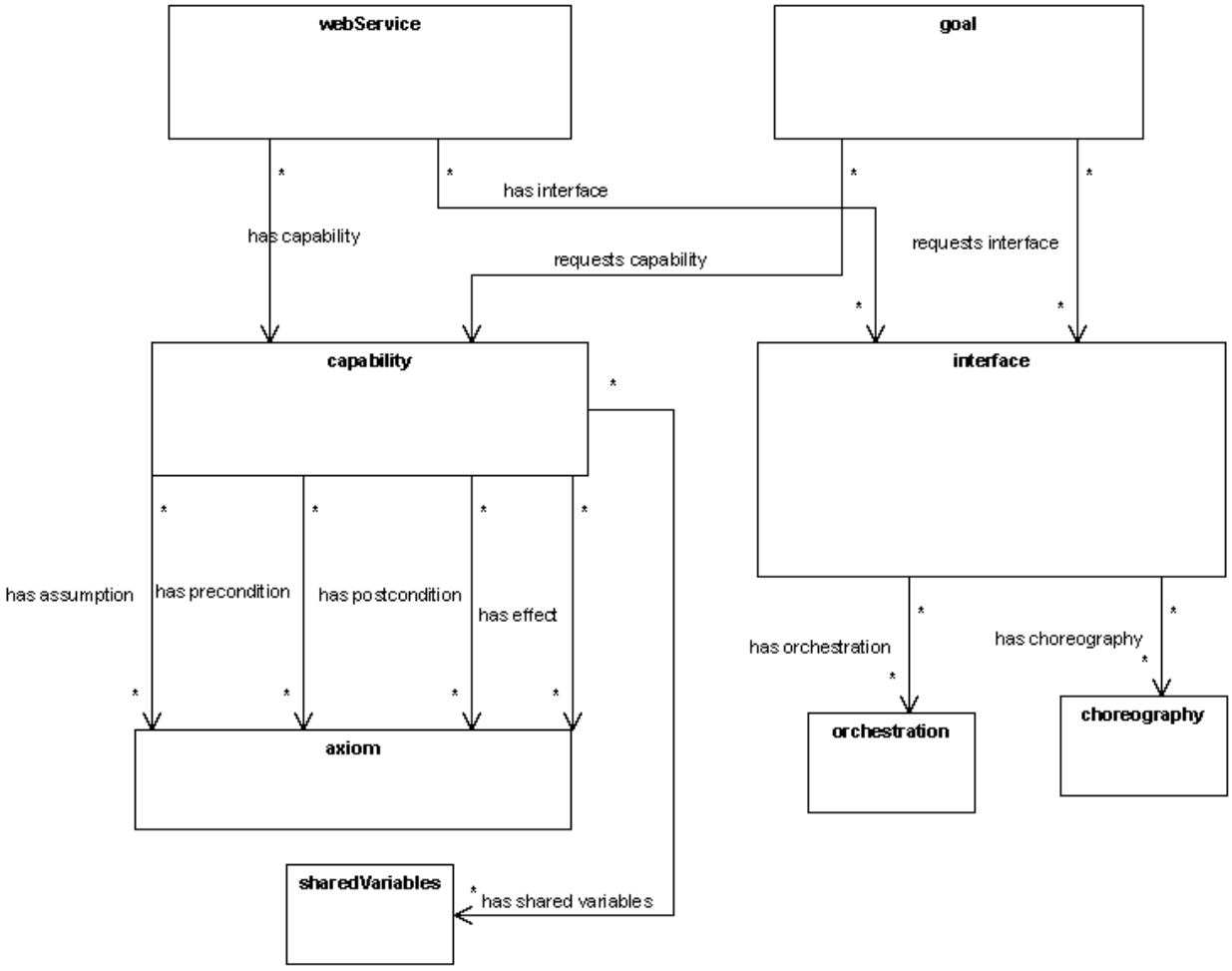
Upper WSMO Elements



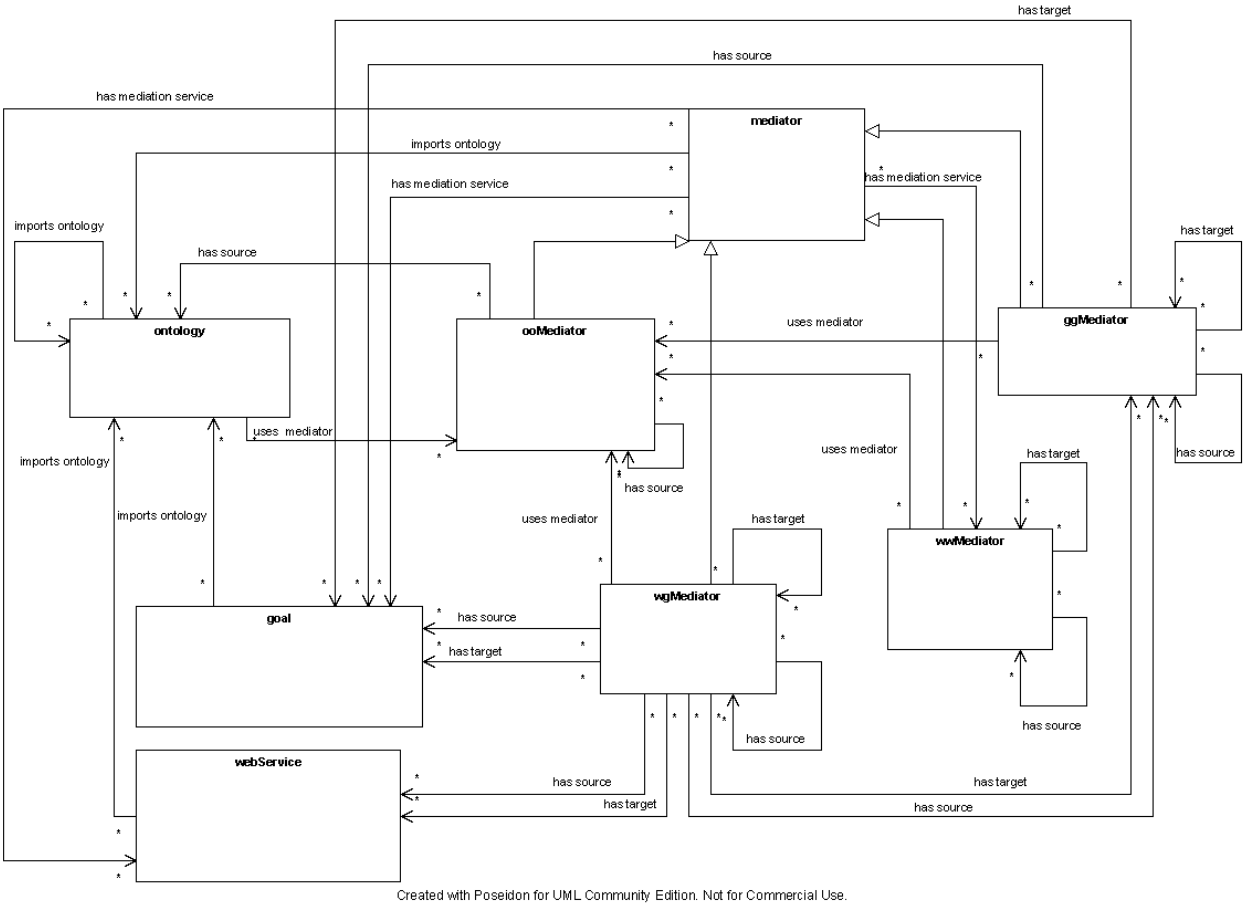
Ontology Related Classes



Goal and Web Service Classes



Mediator Classes



Footnotes:

[1] One could argue that orchestration should not be part of a public interface because it refers to how a Web service is implemented. However, this is a short-sighted view that does not reflect the nature of fully open and flexible eCommerce. Here companies shrink to their core processes which they are really profitable in. All other processes are outsourced and consumed as eServices. eCommerce companies 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 this dinosaurian phase of eCommerce where large companies still exist, `orchestration` may be an important aspect. The orchestration of a Web 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 their 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 where it comes in as a silver bullet in the process. How they manage the difference between the process decomposition at the choreography and the orchestration level is the business intelligence of the service provider.

[2] Note that [Preist, 2004] also distinguishes between a computational entity in general and Web service, where the former does not necessarily have a Web accessible interface. WSMO does not make this distinction.

webmaster

$Date: 2005/04/13 16:59:57 $