



D2v03. Web Service Modeling Ontology - Standard (WSMO - Standard)

WSMO Working Draft 29 March 2004

This version:

<http://www.wsmo.org/2004/d2/v0.3/20040329/>

Latest version:

<http://www.wsmo.org/2004/d2/v0.3/>

Previous version:

<http://www.wsmo.org/2004/d2/v0.3/20040324/>

Editors:

Dumitru Roman
Holger Lausen
Uwe Keller

This document is also available in non-normative [PDF](#) version.

Copyright © 2004 [DERI](#)®, All Rights Reserved. [DERI](#) liability, trademark, document use, and software licensing rules apply.

Table of contents

- [1. Introduction](#)
 - [2. Non functional properties](#)
 - [2.1 Non functional properties - core properties](#)
 - [2.2 Non functional properties - web service specific properties](#)
 - [3. Ontologies](#)
 - [3.1 Axioms](#)
 - [3.2 Concepts](#)
 - [3.3 Relations](#)
 - [3.4 Instances](#)
 - [4. Goals](#)
 - [5. Mediators](#)
 - [6. Web Services](#)
 - [6.1 Capability](#)
 - [6.2 Interfaces](#)
 - [7. Formal Language Specification for WSMO](#)
 - [8. Conclusions and further directions](#)
 - [References](#)
 - [Acknowledgement](#)
 - [Apendix A](#)
-

1. Introduction

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) [Fensel & Bussler, 2002] as a starting point, we refine this framework and develop a formal ontology and a formal language.

The WSMF [Fensel & Bussler, 2002] consists of four different main elements (see [Figure 1](#)): ontologies that provide the terminology used by other elements, goal repositories 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 interoperability problem.

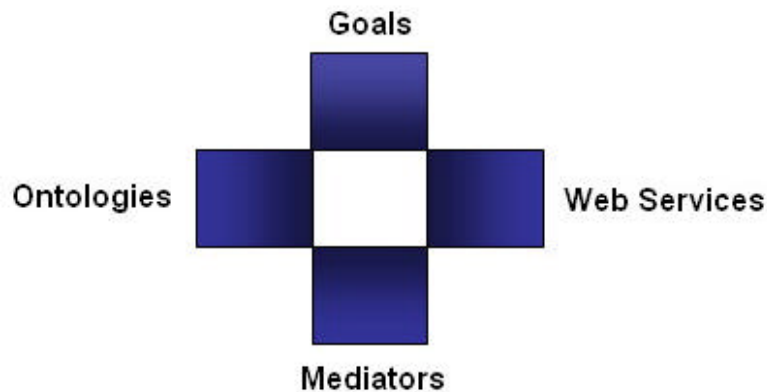


Figure 1. The main elements of WSMF

[Section 2](#) presents the non functional properties of the modeling elements of WSMO. Following the philosophy of WSMF, we further refine in the next sections the ontologies ([Section 3](#)), goals ([Section 4](#)), mediators ([Section 5](#)) and web service ([Section 6](#)). [Section 7](#) presents the formal language we use for specifying WSMO Standard and [Section 8](#) presents our conclusions and further intentions.

2. Non functional properties

Non functional properties are defined as a set of tuples, where each tuple consists of a property and its value constraint. We classify non functional properties in two categories: core properties and web service specific properties.

2.1 Non functional properties - core properties

The core properties are defined globally, meaning that they can be used for all the modeling elements of WSMO Standard. They consist of the [Dublin Core Metadata Element Set \[Weibel et al.\]](#) plus the `version` element:

Listing 1. Non functional properties (core properties) definition

```
nonFunctionalProperties[
title => title
creator => creator
subject => subject
description => description
publisher => publisher
contributor => contributor
date => date
type => type
format => format
identifier => identifier
source => source
language => language
relation => relation
coverage => coverage
rights => rights
version => version
]
```

Title

A name given to an element. Typically, `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 `creator` include a person, an organization, or a service. Typically, the name of a `creator` should be used to indicate the entity.

Subject

A topic of the content of the element. Typically, `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 `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 `publisher` include a person, an organization, or a service. Typically, the name of a `publisher` should be used to indicate the entity.

Contributor

An entity responsible for making contributions to the content of the element. Examples of `contributor` include a person, an organization, or a service. Typically, the name of a `contributor` should be used to indicate the entity.

Date

A date of an event in the lifecycle of the element. Typically, `date` will be associated with the creation or availability of the element.

Type

The nature or genre of the content of the element. The `Type` includes terms describing general categories, functions, genres, or aggregation levels for content.

Format

A physical or digital manifestation of the element. Typically, `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.

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. 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).

Source

A reference to an element from which the present element is derived. The present element may be derived from the `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.

Language

A language of the intellectual content of the element.

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.

Coverage

The extent or scope of the content of the element. Typically, `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).

Rights

Information about rights held in and over the element. Typically, `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.

2.2 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

```
nonFunctionalPropertiesWS :: nonFunctionalProperties
nonFunctionalPropertiesWS [
performance => performance
reliability => reliability
security => security
scalability => scalability
robustness => robustness
accuracy => accuracy
transactional => transactional
trust => trust
financial => financial
networkRelatedQoS => networkRelatedQoS
]
```

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.

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).

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.

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.

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.

Transactional

It represents the transactional properties of the web service.

Trust

It represents the trust worthiness of the service.

Financial

It represents the cost-related properties of a web service.

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.

3. Ontologies

In WSMO Ontologies are the key to link consensual real world semantics intended by humans with computers. Originating from field of philosophy, in AI the term “ontology” refers to a description of a part of the world in a program – a domain of discourse. An important definition of ontology, used by many researchers in the field of ontologies was introduced by Gruber [[Gruber, 1993](#)]: *An ontology is a formal explicit specification of a shared conceptualization*. Respecting this definition, WSMF defines the two essential aspects of ontologies as follows:

- Ontologies define formal semantics for information, consequently allowing information processing by a computer.
- Ontologies define real-world semantics which make it possible to link machine-processable content with meaning for humans based on consensual terminologies.

From this rather conceptual definition we want to extract the essential components which define an ontology. Ontologies define a consensual 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, for instance First-Order Logic.

Each element that belongs to the established terminology, i.e. concepts and relations, can be further constrained semantically by means of a logical constraint that expresses some sort of real-world semantics related to this element.

There are several ways to describe the contents of ontologies by an ontology [1] itself. For now, we decide to keep the model as simple as possible and represent each element of an ontology as simple as possible.

In principle, an ontology constitutes of four main building blocks: concepts, relations, axioms and instances. An ontology is defined as follows [2]:

Listing 3. Ontology definition

```
ontology[
nonFunctionalProperties => nonFunctionalProperties
usedMediators =>> ooMediator
axioms =>> axiomDefinition
concepts =>> conceptDefinition
relations =>> relationDefinition
instances =>> instanceDefinition
]
```

Non functional properties

The non functional properties of an ontology consist of the core properties described in [Section 2.1](#).

Used mediators

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. Imported ontologies allow a modular approach for ontology design. By importing other ontologies, one can make use of concepts and relations defined elsewhere. Nevertheless, when importing an arbitrary ontology, 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, we use ontology mediators (ooMediators) for importing ontologies.

Axioms

The set of axioms that belong to the represented ontology ([Section 3.1](#)).

Concepts

The set of concepts that belong to the represented ontology ([Section 3.2](#)).

Relations

The set of relations that belong to the represented ontology ([Section 3.3](#)).

Instances

The set of instances that belong to the represented ontology ([Section 3.4](#)).

3.1 Axioms

An axiomDefinition is considered to be a logical expression enriched by some extra-logical information.

Listing 4. Axiom definition

```
axiomDefinition[
nonFunctionalProperties => nonFunctionalProperties
defined_by => logicalExpression
]
```

Non functional properties

The `non functional properties` of an axiom consist of the core properties described in [Section 2.1](#).

Defined by

The logical constraint expressed in the formal language underlying the WSMO that represents the actual statement of the axiom.

3.2 Concepts

`Concepts` constitute the basic elements of the consensual terminology for some problem domain. They provide an abstract view on real-existing and artificial artifacts within the addressed domain of discourse.

From a high-level perspective, a concept – described by a concept definition – provides attributes with names and types. It has a name, can be textually described in natural language and might change over time and thus has a version (they are part of the non functional properties of the `concept`).

Furthermore, a concept can have several (possibly none) direct superconcepts as specified by the so-called "is_a"-relation.

When describing the semantics of concepts within some ontology, we favor a uniform and rather general approach: we consider the semantics to be captured by means of a logical expression.

For instance, this allows us to state that some concept represents the union or intersection of two or more other concepts. Consider an ontology on social structures within a human society, and then we can define concepts like "Human-being" or "Female" and accurately describe the semantics of the concept "Granny" as precisely the intersection of the concepts "Human-being", "Female" and "Parent of some parent".

Such modeling styles are commonly used in many *Description Logics* [\[Baader et al., 2003\]](#) and can be found in widely-used ontology languages like *OWL* [\[Dean et al., 2004\]](#) as well.

Hence, we extract the following abstract description for concepts:

Listing 5. Concept definition

```
conceptDefinition :: axiomDefinition
conceptDefinition[
superConcepts =>> conceptDefinition
attributes =>> attributeDefintion
methods =>> methodDefintion
]
```

Superconcepts

There can be a finite number of concepts that serve as direct `superconcepts` for some concept.

In particular, being a subconcept of some other concept means that a concept inherits the signature of this superconcept and the corresponding constraints.

Attributes

Each concept provides a (possibly empty) set of `attributes` that represent named slots for data values and 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 6. Attribute definition

```

attributeDefintion :: axiomDefinition
attributeDefintion[
range => axiomDefinition
]

```

Range

A logical expression constraining the possible values for filling the slot of any instance of a particular concept.

Methods

Besides attributes we also allow a concept to have `methods` [3] that can be invoked on each instance of a concept and in response return some result value.

A method specifies a function that can be invoked on a specific instance of a concept. When invoking the function, one has to specify the values of the parameters, for which the function has to be computed. The specific instance, for which the method is invoked, can be seen as an implicit input parameter of the function, which is not explicitly contained in the set of input parameters. The computed value will then be returned to the invoker.

Listing 7. Method definition

```

methodDefintion :: axiomDefinition
methodDefintion[
range => axiomDefinition
parameters => LIST(parameterDefinition)
]

```

Range

A logical expression constraining the possible values for filling the slot of any instance of a particular concept.

Parameters

A list of the input parameters of the method. Concrete values for these parameters have to be specified when the method will be invoked.

A `parameter` is a named placeholder for some value. This concept is used in the definition of methods as well as in the definition of n-ary relations.

Listing 8. Parameter definition

```

parameterDefinition :: axiomDefinition
parameterDefinition[
domain => axiomDefinition
]

```

Domain

A logical expression constraining the possible values that the parameter can take.

3.3 Relations

`Relations` are used in order to model interdependencies between several concepts (respectively instances of these concepts) with respect to the problem domain.

`Relations` between concepts are more general than simple attributes or properties as for instance in OWL. Mathematically, relationships are simply sets of n-tuples, over the domain of instances of concepts. In popular and commonly used system modeling languages like UML [Fowler, 2003] such concrete tuples are often called links. The underlying semantics of

cardinalities in the case of n-ary relations follows the definition in the UML framework [Rumbaugh et al., 1998].

Relations can be very specific in nature and only applicable in the context of a particular problem domain, but there are also relations that occur frequently when modeling ontologies for different application areas. There are several common properties that modeled relations can provide, e.g. symmetry, transitivity, and reflexivity. Again, for the sake of simplicity we decide not to represent these common properties of relationships currently within the metaontology explicitly but implicitly by means of axioms.

Other dependencies between relationships (for instance subset, intersection, union, difference, and inverse relationship between two or more relations) will be dealt with in the same way.

Listing 9. Relation definition

```
relationDefinition :: axiomDefinition
relationDefinition[
parameters => LIST(parameterDefinition)
]
```

Parameters

A list of `parameter` descriptions specifying each of the concepts that are interrelated.

3.4 Instances

Eventually, within an ontology there might be `instances` defined for some concept. Therefore we have to reflect the “instance_of”-relation that can be given within an ontology specification.

Listing 10. Instance definition

```
instanceDefinition :: axiomDefinition
instanceDefinition[
instanceOf =>> conceptDefinition
attributeValues =>> attributeValueDefinition
]
```

Instance of

We consider the general case, where an instance might be the instance of some (complex) concept which is defined in terms of a logical expression.

Attribute values

A list of attribute values for the instance.

Listing 11. Attribute value definition

```
attributeValueDefinition :: axiomDefinition
attributeValueDefinition[
value => axiomDefinition
]
```

Value

A logical expression defining the values for filling the slot of the instance.

4. Goals

In this section, we introduce the notion of `goals` and define the elements that are used in the description of a goal. Our definition of a goal is the one given in [Fensel & Bussler, 2002]: A

goal specifies the objectives that a client may have when he consults a web service.

In [\[Fensel & Bussler, 2002\]](#), a goal specification consists of two elements, namely:

- Pre-conditions
- Post-conditions

WSMO restricts the definition of goals to Post-conditions. In addition, the Web Services Modeling Ontology introduces `non functional properties`, `used mediators`, and `effects`.

Listing 12. Goal definition

```
goal[
nonFunctionalProperties => nonFunctionalProperties
usedMediators =>> {ooMediator or ggMediator}
postConditions =>> axiomDefintion
effects =>> axiomDefinition
]
```

Non functional properties

The `non functional properties` of a goal consist of the core properties described in the [Section 2.1](#) (where, in this case, an element in the core properties is equivalent to a goal).

Used mediators

By importing ontologies, a goal can make use of concepts and relations defined elsewhere. A goal can import ontologies using ontology mediators (`ooMediators`). A goal may be defined by reusing an already existing goal (e.g. for defining the goal “buy a book for children” one can reuse the already existing goal “buy a book”) or by combining existing goals (e.g. for defining the goal “buy a book from Amazon written by Hemingway” one can reuse and combine the existing goals “buy a book from Amazon” and “buy a book written by Hemingway”). This is achieved by using goal mediators (`ggMediators`).

Post-conditions

`Post-conditions` 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 that can be classified into two different classes (cf. [\[Fensel et al., 2003\]](#)): `refiners` and `bridges` (see [Figure 2](#) below).

`Refiners` can be used to define a new component as a refinement of an existing component. `Refiners` support reuse by minimizing the effort in generating new components from existing ones. In WSMO, we have two `refiners`:

- `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.

In addition to `refiners`, WSMO introduces the notion of `bridges`. `Bridges` support reuse by

enabling two components to interact with each other that would otherwise not cooperate due to interoperability problems in terms of functionality mismatch, data mismatch, protocol, or process mismatch. In WSMO, we have two *bridges*:

- *wgMediators*: mediators that link web service to goals. They explicitly may state the difference (reduction) between the two components and map different vocabularies (through the use of *ooMediators*).
- *wwMediators*: mediators linking two Web Services.

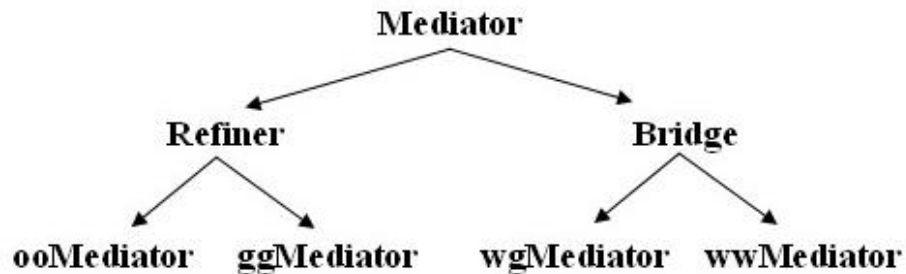


Figure 2. Types of mediators in WSMO

[Figure 3](#) below illustrates the relation between different types of mediators, goals, web services and ontologies.

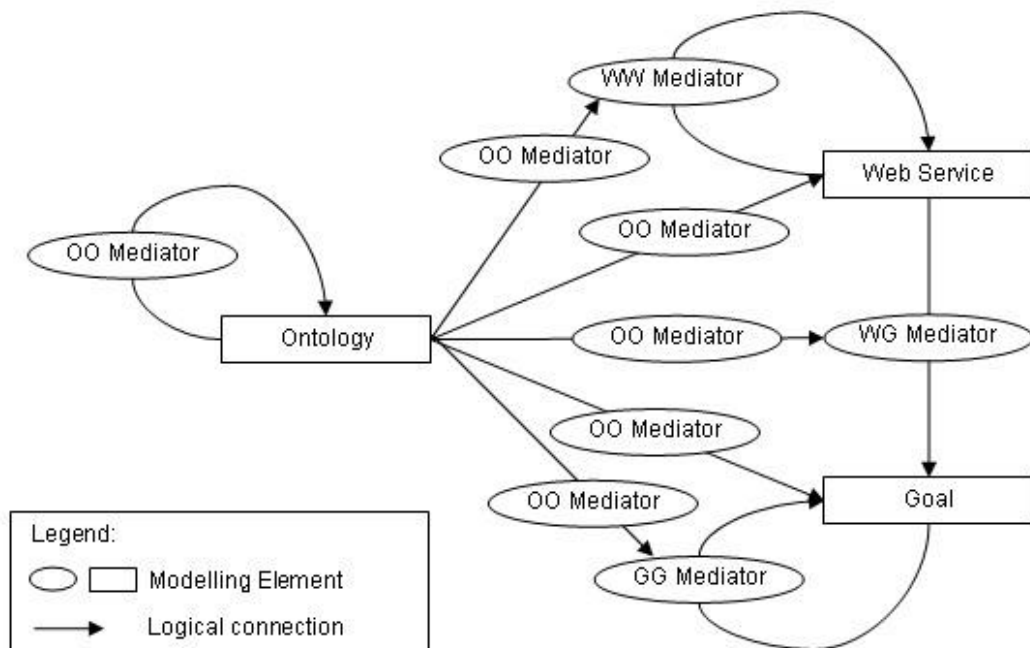


Figure 3. Illustration of mediators in WSMO

The *mediator* is defined as follows:

Listing 13. Mediators definition

```

mediator[
nonFunctionalProperties => nonFunctionalPropertiesWS
sourceComponent =>> (ontology or goal or webService or mediator)
targetComponent =>> (ontology or goal or webService or mediator)
mediationService => (goal or wwMediator)
]

ooMediator :: mediator[
sourceComponent =>> (ontology or ooMediator)
]

ggMediator :: mediator[
sourceComponent => (goal or ggMediator)
targetComponent => (goal or ggMediator)
usedMediators =>> ooMediator
reduction => axiomDefinition
]

wgMediator :: mediator[
sourceComponent => (webService or wgMediator)
targetComponent => (goal or wgMediator)
usedMediators =>> ooMediator
reduction => axiomDefinition
]

wwMediator :: mediator[
sourceComponent => (webService or wwMediator)
targetComponent => (webService or wwMediator)
usedMediators =>> ooMediator
]

```

Non functional properties

The non functional properties of a mediator consist of the core properties described in the [Section 2.1](#) (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 2.2](#) 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.

Source component

The source component defines one of the two logically connected entities.

Target component

The target component defines one of the two logically connected entities.

Mediation Service

The `mediation service` points to a goal that declarative describes the mapping or to a `wwMediator` that links to a web service [4] that actually implements the mapping.

Reduction

A reduction only exists in a `wgMediator` or a `ggMediator`. It describes in a logical formula the differences between the functionality described in the goal and the functionality of the web service (if any) or another goal.

6. Web Services

In this section we identify the concepts needed for describing various aspects of a web

service. From a complexity point of view of the description of a web service, the following properties of a web service are considered: non functional properties, used mediators, capability and interfaces.

Listing 14. Web service definition

```
webService[
nonFunctionalProperties => nonFunctionalPropertiesWS
usedMediators =>> ooMediator
capability => capability
interfaces =>> interface
]
```

Non functional properties

The non functional properties of a web service are described in [Section 2.2](#).

Used mediators

By importing ontologies, a web service can make use of concepts and relations defined elsewhere. A web service can import ontologies using ontology mediators (ooMediators).

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 15. Capability definition

```
capability[
nonFunctionalProperties => nonFunctionalProperties
usedMediators =>> (ooMediator or wgMediator)
preconditions =>> axiomDefinition
postconditions =>> axiomDefinition
assumptions =>> axiomDefinition
effects =>> axiomDefinition
]
```

Non functional properties

The non functional properties of a capability consist of the core properties described in the [Section 2.1](#) (where, in this case, an element in the core properties is equivalent to a capability).

Used mediators

By importing ontologies, a capability can make use of concepts and relations defined elsewhere. A capability can import ontologies using ontology mediators (ooMediators). A capability can be linked to a goal using a wgMediator.

Pre-conditions

Pre-conditions in WSMO describe what a web service expects for enabling it to provide its service. They define conditions over the input.

Post-conditions

Post-conditions in WSMO describe what a web service returns in response to its input. They define the relation between the input and the output.

Assumptions

Assumptions are similar to pre-conditions, however, also reference aspects of the state of the world beyond the actual input.

Effects

`Effects` describe the state of the world after the execution of the service.

The purpose of defining again pre-conditions, post-conditions, assumptions, effects, non functional properties and used mediators in the service capability is to have self-contained web service descriptions, that can be referred or not to the defined goals. In this way, we provide greater flexibility for the use of goals and web services.

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)

With this distinction we provide different decompositions of process/capabilities to the top (service requester) and to the bottom (other service providers). 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. 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 choreography 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 he 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. The firewall that is depicted in [Figure 4](#) between `choreography` and `orchestration` refers to the actual implementation of the web service and its underlying process mediation aspect.

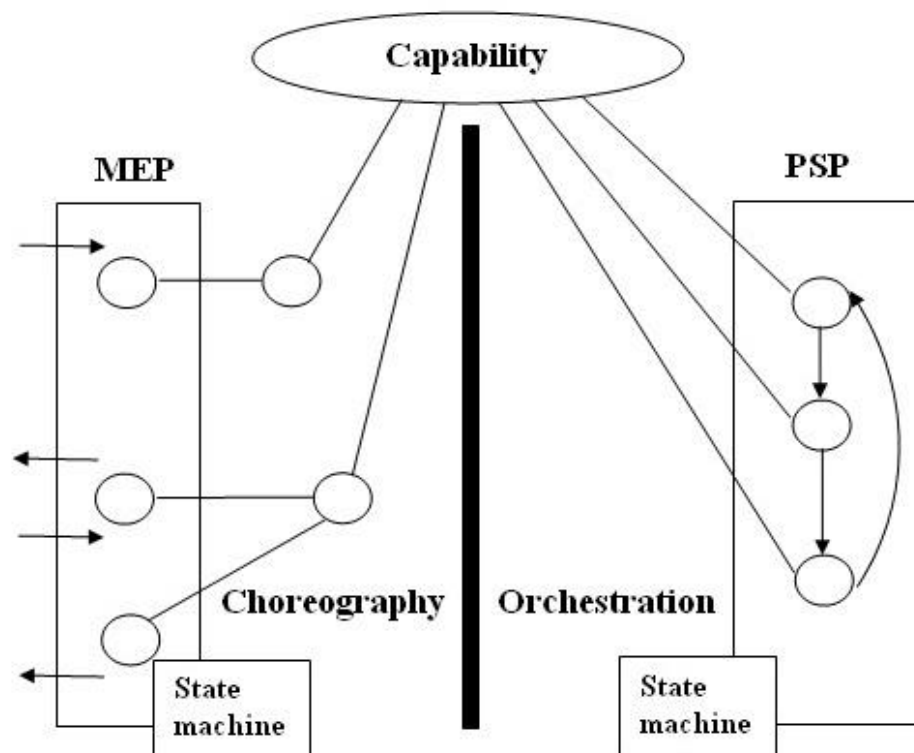


Figure 4. Choreography and Orchestration.

An interface is defined by the following properties:

Listing 16. Interface definition

```
interface[
nonFunctionalProperties => nonFunctionalProperties
usedMediators =>> ooMediator
choreography => instantiatedCMEP
orchestration => instantiatedPSP
]
```

Non functional parameters

The non functional properties of an interface consist of the core properties described in the [Section 2.1](#).

Used mediators

By importing ontologies, an interface can make use of concepts and relations defined elsewhere. An interface can import ontologies using ontology mediators (ooMediators).

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). A choreography achieves this by instantiating a Message Exchange Pattern (MEP) for the given capability of a web service. A message exchange pattern specifies a sequence of conditional speech acts that either send or receive communication symbols (messages in our case). In between the communication ask, non-specified actions are defined. State-less MEPs model stimulus-response patterns (i.e. the behavioral point of view) and state-based MEPs model conversations (i.e. the modern cognitive science point of view). A choreography instantiates a generic communication pattern. That means:

- A choreography decomposes the capability of a web service capability into sub-capabilities. These sub-capabilities are used to define the generic activities in a MEP.
- Conditions are specified.

- The I/Os of these sub-capabilities define the message formats of an instantiated MEP.
- The collection of send and received messages define the state description of an instantiated MEP.

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).

An `orchestration` describes how the service works by instantiating a `Problem Solving Pattern (PSP)` for the given capability of a web service. A `problem solving pattern` specifies a sequence of conditional activities. Again, a description can be state-less or state-based. An `orchestration` instantiates a generic cooperation pattern. That means:

- An `orchestration` decomposes the capability of a web service into sub-capabilities. These sub-capabilities are used to define the activities in a PSP. It specifies a set of proxies that the service uses in order to fulfill its functionality. A **proxy** is defined as being either a goal or a `wwMediator`. Proxies are used when a web service invokes other web services in order to provide its service. Each time a web service needs to be invoked, a proxy needs to be declared (by either declaring a goal or linking it to a `wwMediator`). This way both dynamic (on the fly) composition (by declaring proxies consisting of goals descriptions) and static composition (by linking proxies to `wwMediators`) are supported.
- Conditions are specified.
- The collection of I/Os of the sub-capabilities define the state description of an instantiated PSP.

The distinction between `choreography` and `orchestration` 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 (Problem Solving Pattern) and
- the way of interacting with them

are specified in the orchestration of the service.

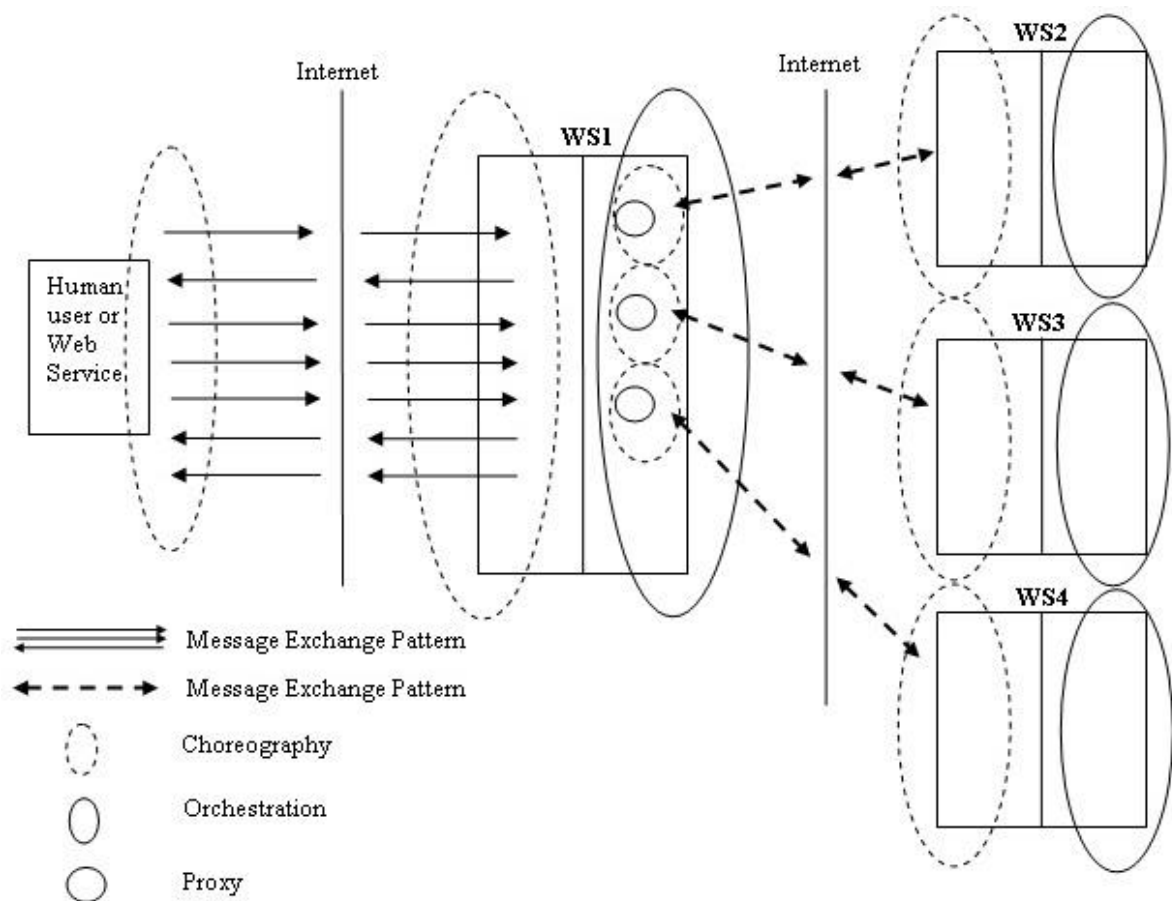


Figure 5. Choreography vs. Orchestration.

Figure 5 depicts this distinction between choreography and orchestration. WS1, viewed as a provider, describes its choreography such that the human user or the web service (the left most part in the figure) can either generate (in case it doesn't have any choreography described), or adapt its choreography to the choreography of WS1 (if it already has a choreography defined), thus being able to converse with the service.

Note that the actual conversation is taking place only between choreographies.

WS1, viewed as a requester, might need to use other services in order to fulfill its functionality. Each service is represented in the orchestration by a proxy, which can be either a goal (in order to allow dynamic composition of services) or a web service (in order to allow static composition of services). The way these proxies are connected is part of the orchestration.

For each proxy in the orchestration a choreography can be associated in order to describe the actual conversation with the service the proxy represents (i.e. WS2, WS3, WS4 in the figure). Usually for proxies who are goals the choreography will be automatically generated according to the choreography of the service which will be dynamically binded at runtime, and for proxies who are services a wwMediator will be generated in order to adapt both choreographies.

The choreography in WSMO is further defined in [Deliverable 14: Choreography in WSMO\[5\]](#).

The orchestration in WSMO is further defined in [Deliverable 15: Orchestration in WSMO\[5\]](#).

7. Formal Language Specification for WSMO

As shown in the previous sections, many properties of goals, mediators, ontologies and web services are considered to be axioms, defined by logical expressions. In order to be as specific as possible, we consider these logical expressions as being described in F-Logic [[Kifer et al., 1995](#)], by complex formulas.

F-Logic combines the advantages of conceptual high-level approaches typical for frame-based language and the expressiveness, the compact syntax, and the well defined semantics from logics. Features of F-Logic include: object identity, methods, classes, signatures, inheritance and rules.

The reasons for which we choose F-Logic to represent logical expressions are:

- it provides a standard model theory
- it is a full first order logic language
- it provides second order syntax while staying in the first order logic semantics
- it has a minimal model semantics
- implemented inference engines are already available

For a detailed description of F-Logic refer to [[Kifer et al., 1995](#)]. Furthermore, an evaluation of F-Logic and comparison to other languages is given in [[Keller et al., to appear](#)].

Finally, F-Logic defines a natural framework for extending description logics such as OWL (see [[Balaban, 1995](#)]) and is therefore in line with ongoing initiatives on extending languages such as OWL by rule capabilities.

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). Further versions of this document will contain more detailed descriptions of some concepts presented here as well as refinements that add concepts to the current ontology.

WSMO is centered on the underlying technology to be described. The next step is to define a layer on top in terms of the application that is supported by it: fully flexible eCommerce and eWork.

References

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

[[Balaban, 1995](#)] M. Balaban: The F-Logic Approach for Description Languages, *Annals of Mathematics and Artificial Intelligence*, 15, pp. 19-60, 1995.

[[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/>.

[[Fensel & Bussler, 2002](#)] D. Fensel and C. Bussler: *The Web Service Modeling Framework WSMF*, Electronic Commerce Research and Applications, 1(2), 2002.

[[Fensel et al., 2003](#)] D. Fensel, E. Motta, F. van Harmelen, V.R. Benjamins, S. Decker, M.

Gaspari, R. Groenboom, W. Grosso, M. Musen, E. Plaza, G. Schreiber, R. Studer, and B. Wielinga: The Unified Problem-solving Method Development Language UPML, *Knowledge and Information Systems(KAIS) journal*, Vol. 5, No. 1, 2003.

[Fowler, 2003] M. Fowler: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 3rd edition, 2003.

[Gruber, 1993] T. Gruber: A translation approach to portable ontology specifications, *Knowledge Acquisition*, 5:199-220, 1993.

[Keller et al., to appear] U. Keller, A. Polleres, R. Lara, and Y. Ding: *Language Evaluation and Comparison*, to appear.

[Kifer et al., 1995] M. Kifer, G. Lausen, and James Wu: Logical foundations of object oriented and frame-based languages. *Journal of the ACM*, 42(4):741-843, 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>.)

[Rumbaugh et al., 1998] J. Rumbaugh, I. Jacobson, and G. Booch: *The Unified Modeling Language Reference Manual*, Object Technology Series, Addison-Wesley, 1998.

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

Acknowledgement

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

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

Apendix A. Flora2 Syntax for WSMO

```
nonFunctionalProperty
[
  type *=> string,
  rights *=> string,
  subject *=> string,
  format *=> string,
  coverage *=> string,
  creator *=> string,
  relation *=> string,
  contributor *=> string,
  publisher *=> string,
  source *=> string,
  date *=> string,
  language *=> string,
  version *=> string,
  title *=> string,
  description *=> string
].

nonFunctionalPropertyWS::nonFunctionalProperty
[
```

```

    identifier *=> string,
    financial *=> string,
    security *=> string,
    networkRelatedQoS *=> string,
    transactional *=> string,
    performance *=> string,
    scalability *=> string,
    reliability *=> string,
    robustness *=> string,
    accuracy *=> string,
    trust *=> string
].

ontology
[
    identifier => string,
    nonFunctionalProperties => nonFunctionalProperty,
    usedMediators =>> ooMediator,
    conceptDefinitions =>> conceptDefinition,
    instances =>> instance,
    axioms =>> axiomDefinition,
    relations =>> relationDefinition
].

webService
[
    identifier => string,
    interfaces =>> interface,
    capability => capability,
    nonFunctionalProperties => nonFunctionalPropertyWS,
    usedMediators =>> ooMediator
].

goal
[
    identifier => string,
    postconditions =>> axiomDefinition,
    nonFunctionalProperties => nonFunctionalProperty,
    effects =>> axiomDefinition,
    usedMediators =>> oo_or_ggMediator
].

mediator
[
    identifier *=> string,
    nonFunctionalProperties *=> nonFunctionalPropertiesWS,
    sourceComponent *=>> goal_or_mediator_or_ontology_or_webService,
    targetComponent *=>> goal_or_mediator_or_ontology_or_webService,
    mediationService *=> goal_or_wgMediator
].

ooMediator::mediator
[
    sourceComponent =>> ontology_or_ooMediator
].

ggMediator::mediator
[
    sourceComponent =>> goal_or_ggMediator,
    targetComponent =>> goal_or_ggMediator,
    usedMediators =>> ooMediator,
    reduction => axiomDefinition
].

wgMediator::mediator

```

```

[
  sourceComponent =>> webService_or_wgMediator,
  targetComponent =>> goal_or_wgMediator,
  usedMediators =>> ooMediator,
  reduction => axiomDefinition
].

wwMediator::mediator
[
  sourceComponent =>> webService_or_wwMediator,
  targetComponent =>> webService_or_wwMediator,
  usedMediators =>> ooMediator
].

axiomDefinition
[
  identifier *=> string,
  definedBy *=> logicalExpression,
  nonFunctionalProperties *=> nonFunctionalProperty
].

instance::axiomDefintion
[
  instanceOf =>> conceptDefinition,
  attributeValues =>> attributeValueDefinition
].

attributeValueDefinition::axiomDefinition
[
  value => axiomDefinition
].

relationDefinition :: axiomDefinition
[
  parameters =>> parameterDefinition
].

parameterDefinition :: axiomDefinition
[
  domain => axiomDefinition
].

conceptDefinition :: axiomDefinition
[
  superConcepts =>> conceptDefinition,
  attributes =>> attributeDefintion,
  methods =>> methodDefintion
].

methodDefintion :: axiomDefinition
[
  range => axiomDefinition,
  parameters =>> parameterDefinition
].

capability
[
  identifier => string,
  nonFunctionalProperties => nonFunctionalProperties,
  usedMediators =>> ooMediator_or_wgMediator,
  preconditions =>> axiomDefinition,
  postconditions =>> axiomDefinition,
  assumptions =>> axiomDefinition,
  effects =>> axiomDefinition
].

```

```

interface
[
    identifier => string,
    nonFunctionalProperties => nonFunctionalProperties,
    usedMediators =>> ooMediator,
    choreography => instantiatedCMEP,
    orchestration => instantiatedPSP
].

% goal_or_mediator_or_ontology_or_webService
goal::goal_or_mediator_or_ontology_or_webService.
mediator::goal_or_mediator_or_ontology_or_webService.
ontology::goal_or_mediator_or_ontology_or_webService.
webService::goal_or_mediator_or_ontology_or_webService.
goal_or_mediator_or_ontology_or_webService.

%ontology_or_ooMediator
ontology::ontology_or_ooMediator.
ooMediator::ontology_or_ooMediator.
ontology_or_ooMediator.

%goal_or_wwMediator
goal::goal_or_wwMediator.
wwMediator::goal_or_wwMediator.
goal_or_wwMediator.

% ooMediator_or_wgMediator
ooMediator::oo_or_ggMediator.
ggMediator::oo_or_ggMediator.
oo_or_ggMediator::mediator.

%goal_or_ggMediator
goal::goal_or_ggMediator.
ggMediator::goal_or_ggMediator.
goal_or_ggMediator.

%webService_or_wwMediator
webService::webService_or_wwMediator.
wwMediator::webService_or_wwMediator.
webService_or_wwMediator.

%webService_or_wgMediator
webService::webService_or_wgMediator.
wgMediator::webService_or_wgMediator.
webService_or_wgMediator.

%goal_or_wgMediator
goal::goal_or_wgMediator.
wgMediator::goal_or_wgMediator.
goal_or_wgMediator.

```

[1] Such an ontology actually represents a metaontology.

[2] The notation used for defining the elements of WSMO is based on F-Logic syntax.

[3] It's obvious, that this additional modeling element for describing ontologies only makes sense, if the logical framework underlying WSMO allows one to define and use methods within axioms. Since WSMO is principally based on F-Logic, this is clearly the case.

[4] Notice that the capability of this web service will also provide a declarative description of this mapping; however, in addition a link to an implementation of the mapping is provided, too.

[5] The grounding of a web service (i.e. the mapping from the abstract specification to a concrete specification of the elements that describe a web service, which are required for interacting with the service) will be specified in the choreography and orchestration, as they are the only elements where grounding is needed.

[webmaster](#)