



D29v0.1 WSMO Mediators

WSMO Working Draft 25 November 2005

This version:

<http://www.wsmo.org/TR/d29/v0.1/20051125/>

Latest version:

<http://www.wsmo.org/TR/d29/>

Previous version:

<http://www.wsmo.org/TR/d29/v0.1/20051111/>

Editor:

Adrian Mocan
Emilia Cimpian
Michael Stollberg
Francois Scharffe
James Scicluna

Authors* and Responsibilities:

Emilia Cimpian (section 4: WG Mediators)
Adrian Mocan (section 2: OO Mediators, Introduction)
Francois Scharffe (conclusion)
James Scicluna (section 5: WW Mediators)
Michael Stollberg (Introduction, section 3: GG Mediators, section 5: WW Mediators)

*: Alphabetical order

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

[1.1. Mediation in WSMO - Aims and Approach](#)

[1.2. Related Work](#)

2. OO Mediators

[2.1. Aims and Usage](#)

[2.2. OO Mediator Definition](#)

[2.3 Related Mediation Techniques](#)

[2.3.1. Requirements](#)

2.3.2. Used Techniques

3. GG Mediators

3.1. Aims and Usage

3.2. GG Mediator Definition

3.3. Related Mediation Techniques

3.3.1. Requirements

3.3.2. Used Techniques

4. WG Mediators

4.1. Aims and Usage

4.2. WG Mediator Definition

4.3. Related Mediation Techniques

4.3.1. Requirements

4.3.2. Used Techniques

5. WW Mediators

6. Conclusions

References

Appendix A. Ontology Mapping Language

Acknowledgements

1. Introduction

This document provides a more detailed specification of the concept of Mediators as a top level element of the Web Service Modeling Ontology WSMO. Mediation is concerned with handling heterogeneity by resolving possibly occurring mismatches between resources that whose interoperability would be useful but is not given a priori. As heterogeneity naturally arises in open and distributed environments, and thus in the application areas of Semantic Web Services, WSMO identifies Mediators as a core element of Semantic Web Services. The aim is to define specification, usage and mediation techniques of WSMO Mediators as an extension of the definition provided in the WSMO specification [[Roman et al., 2005](#)].

The document is structured as follows. The remainder of Section 1 outlines the aim and approach for mediation in WSMO, Sections 2 to 5 specify the distinct types of Mediators in detail, and Section 6 concludes the document.

1.1 Mediation in WSMO - Aims and Approach

Heterogeneity is an inherent characteristic of open and distributed environments like the Internet that hampers interoperability and thus automated Web service usage. A major merit of the Semantic Web and Semantic Web services is that all information and resources carry unambiguous semantic descriptions. This allows development of general purpose mediation techniques and infrastructures that work on declarative, semantic resource descriptions. Consequently, WSMO aims at providing techniques and an infrastructure for handling all kinds of heterogeneity that potentially occur within Semantic Web services. The approach taken in WSMO for realizing an integrated mediation framework for Semantic Web services is explained in the following.

In order to tackle heterogeneity handling as a major issue within the Semantic Web and Semantic Web services, WSMO defines the concept of Mediators as a top level notion in order to support mediation-orientated architecture for Semantic Web services. Implementing the WSMO design concept of strong decoupling and strong mediation, the WSMO mediation framework presented in this document is comprised of a general structure definition of mediators as architectural components and a typology of different mediators along with infrastructural correlations between them.

A WSMO mediator connects heterogeneous components and resolves mismatches between them. The general structure of WSMO mediators is shown in Figure 1, stating:

- a Mediator can have one or several *source components*, and one or several *target components*. These denote heterogeneous resources; the heterogeneities are resolved by the mediator ;
- therefore, so-called *mediation definitions* define how mismatches are resolved in an appropriate mediation definition language, and
- a *mediation service*, a Web service that is capable of executing the mediation definition. The link to the used mediation service can either be directly or indirectly: indirect mediator service usage can be specified via a goal, including a discovery process for an adequate service to be used, or defined via another mediator that resolves mismatches between the mediator and the mediation service.

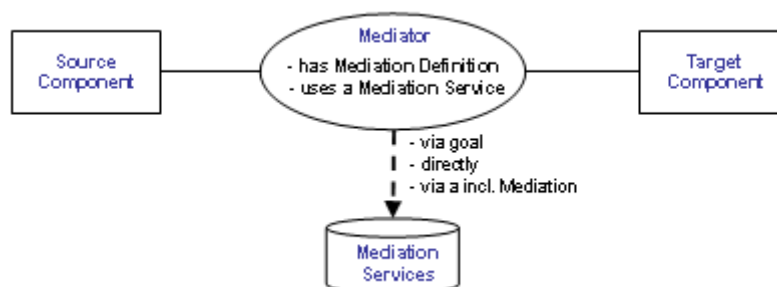


Figure 1: WSMO Mediator Structure

Heterogeneities that might hamper Web service providers and requesters interacting successfully can arise on different levels, e.g. between terminologies or representation formats used by distinct entities. For resolving mismatches that potentially occur on each of these levels, corresponding mediation techniques are needed. WSMO understands its top level elements - Ontologies, Goals, Web Services, and Mediators - as the core elements of Semantic Web service technology. In consequence, WSMO distinguishes four mediator types that connect related WSMO elements and resolve mismatches between them. The different mediators are named by prefixes, denoting WSMO elements as the respective source and target components, and use respective mediation techniques for resolving mismatches that can potentially occur between the source and target components. Namely, these are *OO Mediators* that connect ontologies and resolve terminology as well as representation and protocol mismatches, *GG Mediators* that connect Goals, *WG Mediators* that connect Web services and Goals, and *WW Mediators* that connect Web services. Understanding the WSMO top level elements as the core elements for Semantic Web services, these mediator types specifically allow defining the

heterogeneity handling with respect to the mismatches that might occur between these elements.

Figure 2 gives an overview of the WSMO Mediator types, denoting their source and target elements and the mediation techniques used as well as correlations between them with further explanations below.

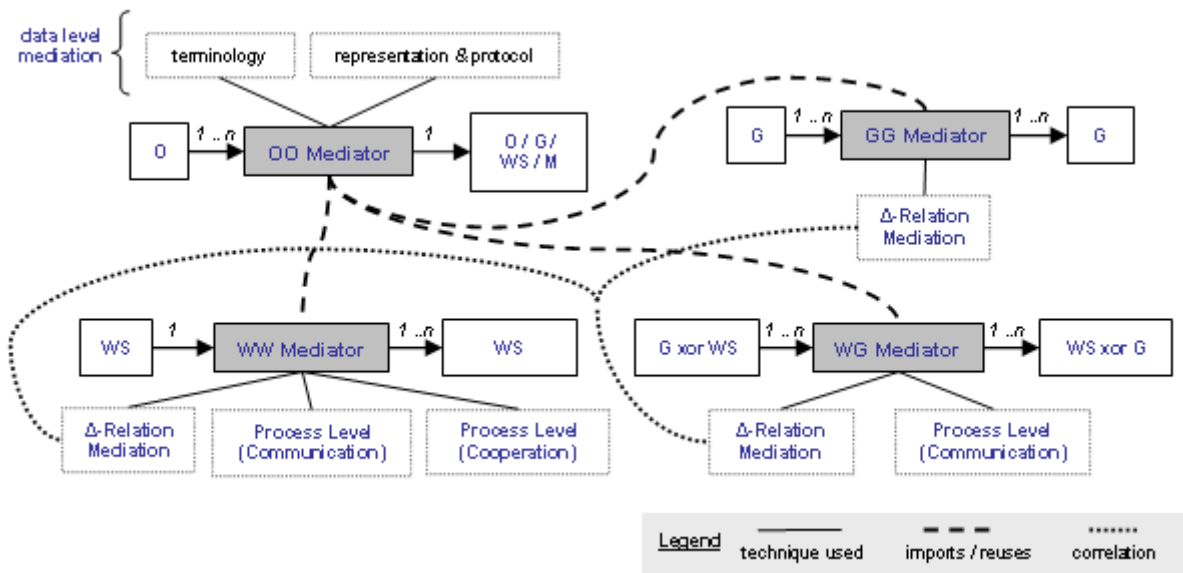


Figure 2: WSMO Mediator Types and Correlations

- **OO Mediators** are concerned with data level mediation, i.e. resolving terminological mismatches between elements. Its source elements are ontologies or ooMediators, and its target component can be any WSMO top level element. The related mediation technique is **data mediation**, which in ontology-based frameworks like WSMO refers to ontology integration (i.e. mapping, merging, and alignment of ontologies) as well as to "lifting / lowering" from syntactical to the ontological level and vice versa. [Section 2](#) addresses WSMO OO Mediators in more detail.
- **GG Mediators** explicitly state the relation between Goals and resolve possibly occurring mismatches between them. The source elements are one or more Goals, and the target component is one Goal. The mediation techniques required for GG Mediators are first **data mediation** by usage of OO Mediators, and so-called **Δ relation mediation** that allows describing precisely the logical relationship between source and target goals. [Section 3](#) addresses WSMO GG Mediators in more detail.
- **WG Mediators** explicitly state the relation between Web Services and Goals and resolve possibly occurring mismatches between them in order to provide auxiliary support for Web Service discovery. WG Mediators can be defined in two directions: either, the source elements are one or more Web Services and the target is a Goal, or the other way around. The mediation techniques for WG Mediators are first **data mediation** by usage of OO Mediators, **Δ relation mediation** for expressing the logical relationship between the source and target elements, and **process mediation for communication** for resolving mismatches between the Choreography Interface definitions of the source and target components. [Section 4](#) addresses WSMO WG Mediators in more detail.

- **WW Mediators** resolve mismatches between Web Services that hamper these from automated interaction. The source component of a WW Mediator is the Web Service that aggregates other Web Services in its Orchestration, and the target component is one of the aggregated Web Services (i.e. the Orchestrator can use several WW Mediators, one for each Web Service aggregated in the Orchestration where mismatches need to be resolved). The related mediation techniques are first **data mediation** by usage of OO Mediators, secondly **Δ relation mediation** for stating logical differences between source and target Web Services, and **process mediation for communication and for coordination**. [Section 5](#) addresses WSMO WW Mediators in more detail.

With respect to an integrated mediation architecture for Semantic Web services, the following architectural correlations hold as indicated in Figure 1:

1. All data level mismatches are handled by OO Mediators. Terminological as well as representation or protocol mismatches that occur in GG, WG, and WW Mediators are handled by importing and re-using OO Mediators
2. In case that the same Goals and Web services are connected in GG, WG, and WW Mediators, specific logical correlations exists between the Δ relations defined in the respective mediators
3. Process level mediation techniques for communication as well as for cooperation are third party facilities capable of establishing behavioral interoperability between Goals and Web services.

A formal specification of the WSMO mediation architecture is considered as future work. The following sections define each mediator type in more detail. Thereby, we subsequently introduce the identified mediation techniques for Semantic Web services.

1.2 Related Work

While related work on the distinct mediation techniques is discussed in the subsequent sections, we briefly examine work related to mediation architectures.

With respect to heterogeneity being identified as a major issue for future IT systems, [\[Wiederhold, 1994\]](#) propagated mediator-orientated architectures in the early 1990ies. A mediator is a special kind of software component capable of dynamically handling heterogeneities that hamper functional components from successful interoperation. The aim is to develop general purpose mediation techniques that work on abstract, semantic level independent of concrete application domains. Therefore, a mediator is understood as an entity capable of establishing interoperability of resources that are not compatible a priori by resolving mismatches between them at runtime. The aspired approach for mediation relies on declarative description of resources whereupon mechanisms for resolving mismatches work on a structural, semantic level, in order to allow defining of generic, domain independent mediation facilities as well as reuse of mediators. The WSMO mediation framework follows this idea.

Concerning the needs for mediation within Semantic Web services, the Web Service Modeling Framework WSMF - the conceptual basis of WSMO - distinguishes three

levels of mediation [[Fensel and Bussler, 2002](#)]. (1) **Data Level Mediation** - mediation between heterogeneous data sources; within ontology-based frameworks like WSMO, this is mainly concerned with ontology integration, (2) **Protocol Level Mediation** - mediation between heterogeneous communication protocols, i.e. translation between technical transfer protocols (e.g. SOAP, HTTP, etc.), and (3) **Process Level Mediation** - mediation between heterogeneous business processes; in WSMO, this is concerned with mismatch handling on behavioral Web Service Interface descriptions for information interchange, communication, and cooperation between Web services and clients. The mediation framework for presented in this paper covers all of these levels, and introduces another level: **Δ-Relation Mediation** for explicitly describing the logical relationship between functional descriptions of goals and Web services in order to enable efficient resource management.

An early approach for realizing a mediation technology that follows Wiederhold's propagation has been presented in the MedMaker project in the mid 1990ies [[Papakonstantinou et al., 1996](#)]. The approach is based on a proprietary, not ontology-based description language for resources called the Object Exchange Model (OEM), and a Mediator Specification Language (MSL), which both are defined as FOL languages. The latter is used for specifying rules that integrate heterogeneous OEM resource descriptions, thereby enabling information interchange between heterogeneous resources. The referenced paper further presents a system implementation Mediator Specification Interpreter (MSI) that is capable of reading and executing MSL specifications. This work can be seen as a predecessor of data level mediation as realized in OO Mediators (see [Section 2](#)). Therein, OEM refers to ontologies, respectively WSMO descriptions of goals and Web Services, while MSL refers to the ontology mapping language as specified in [Appendix A](#).

A more recent approach concerned with the formal specification of mediators as software components is presented in [[Barros and Borger, 2005](#)]. The authors propose eight basic mediation patterns, four for bilateral communication and four for the multilateral mediation patterns and further define combinations and refinements of the basic patterns. However, all these basic patterns and their combinations/refinements are defined using hard-coded Abstract State Machines [[Borger, 1998](#)] and before-hand defined predicates, obtaining in this way an inflexible, rigid model. In our approach we would like to be more flexible, for allowing easily extensions of the addressed patterns.

2. OO Mediators

OO Mediators (or *ooMediators*) are fundamental components of the mediation mechanism proposed by WSMO. They represent bridging entities between the ontologies used to semantically describe all the other WSMO entities. *ooMediators* can be used by any WSMO element (including mediators) when the ontologies needed in the modelling of particular semantic descriptions contain overlapping (maybe even conflicting) aspects. In this section we discuss the aims of the *ooMediators* and how they can be used, we go in detail through the definition of an *ooMediator* and we continue with aspects of the underlying mediation techniques.

The following gives a general overview on the aims and usages of an *ooMediator* (subsection [2.1](#)), followed by a detailed analyses of its definition (subsection [2.2](#)).

The last subsection (subsection [2.3](#)) presents some aspects related to the mediation techniques behind an *ooMediator*.

2.1 Aims and Usage

Conforming to WSMO, all aspects related to Semantic Web Services have to be semantically described using ontologies. A creator of such descriptions has the choice of creating their own ontology or to reuse already existing ones. As the ontologies are "shared conceptualizations", reusing ontologies should be the first choice in any modelling process and WSMO directly supports this by the *importsOntology* statement. Sometimes it might be necessary to refer to multiple ontologies that can describe overlapping domains using different concepts and even conflicting models.

In the next section we describe in details the structure of an *ooMediator*, emphasizing aspects that can be useful in both the creation and the reuse of *ooMediators*.

2.2 OO Mediator Definition

Applying the inheritance principles we derive the following structure for an *ooMediator*:

Listing 1. OO Mediators definition

```

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

```

In the rest of this section we will expand and particularize the definition of each of its constituents presented in [\[Roman et al., 2005\]](#). By this we intend to capture the specific characteristics of the *ooMediators* and to provide the prerequisites for their creation and usage.

Non-Functional Properties (*hasNonFunctionalProperties*)

The non-functional properties of an *ooMediator* remain identical with the general definition of non-function properties for mediators as described in [\[Roman et al., 2005\]](#). In addition, we recommend the usage of another non-functional property that can play an important role in discovery and selection of *ooMediators*:

```
usedMappingLanguage type mappingLanguage
```

By using this non-functional property the designer of the *ooMediator* can offer an insight into the internal mechanism behind the mediation service.

Imported Ontologies (*importsOntology*)

The *importsOntology* statement points to the ontology that describes the terms used in defining the *ooMediator*. For example the mapping language referred by the above

non-functional property can be described in more details in the imported ontology. As we don't have the *usesMediator* statement in an *ooMediator* it is assumed that the imported ontologies are free of heterogeneity problems.

Source (*hasSource*)

An *ooMediator* can have as a source either an *ontology* or an *ooMediator*. In the first case this indicates that the mediation process will be applied to entities that can be found in the name space of this ontology.

The second case is more interesting having as source component an *ooMediator*. Let's consider the following example:

Listing 2. Example of *ooMediator* as source component for another *ooMediator*

```
ooMediator _"http://example.org/ooMediators/secondMediatorExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  hasSource _"http://example.org/ooMediators/firstMediatorExample"
  hasTarget _"http://example.org/ontologies/secondTargetOntology"
  ...

ooMediator _"http://example.org/ooMediators/firstMediatorExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  hasSource _"http://example.org/ontologies/firstSourceOntology"
  hasTarget _"http://example.org/ontologies/firstMediatorExample"
  ...
```

In this case, the role of the source for the *secondMediatorExample* will be played by the target of the *firstMediatorExample*. This means for this example, that the mediation process will be applied to entities that can be found in the name space of *firstMediatorExample*. *firstMediatorExample* might be a syntactic mediator having the role of converting an ontology (e.g. *firstSourceOntology*) from one representation language into another.

As a general remark, the source of an *ooMediator* should never be referred directly in a WSMO entity that uses that particular *ooMediator*. The reason is that an *ooMediator* is used to make that source entity available in a heterogeneity-free and consistent way in the component using the *ooMediator*. By going around the *ooMediator* and directly referring concepts, relations, individuals etc. from the source component the description that is currently built might become inconsistent.

Target (*hasTarget*)

There are four possible cases with respect to the types of entities allowed as target components for an *ooMediator*:

- *ontology* – It is the most straightforward case of the four, and implies that the result of mediation is/contains terms accessible through the name space of this ontology. For example if the *ooMediator* is used in an instance transformation scenario, the transformed instances will be expressed in terms

of the target ontology and accessible through the name space of that ontology. If the ooMediator is used in a query rewriting scenario, each of the ontological entities that appear in the rewritten query is accessible through the name space of this ontology.

- *goal* – An ooMediator with a goal as a target component, resolves the heterogeneity problems between its source ontology and the ontologies imported by the goal. According to [Roman et al., 2005] a goal has the following definition:

Listing 3. Goal definition from [Roman et al., 2005]

```

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

```

- Important for these discussions are the *importsOntology* and *usesMediator* statements. Our ooMediator has to be specified in the *usesMediator* statement while the *importsOntology* has to point to the source ontology of this ooMediator. An example of these would be:

Listing 4. Example of a goal as target component for an ooMediator

```

ooMediator _"http://example.org/ooMediators/firstMediatorExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  hasSource _"http://example.org/ontologies/firstOntology"
  hasTarget _"http://example.org/goals/goalExample"
  ...

goal _"http://example.org/goals/goalExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  usesMediator _"http://example.org/ooMediators/firstMediatorExample"
  importsOntology {_"http://example.org/ontologies/firstOntology",
    ...}
  ...

```

- The mediator guarantees that the problems of heterogeneity between the source ontology (in our example *firstOntology*) and all the imported ontologies are solved. Please note that this mechanism can be used to determine which mediator was used to import a particular ontology when multiple ontologies are imported and multiple ooMediators are used.
- *Web Service* – This is a similar case as the one presented above when substituting the goal with a Web Service.
- *ooMediator* – In the case of *ooMediator* there is one important aspect we have to consider compared to the *Web Services* and *Goals*: an *ooMediator* doesn't have a *usesMediator* statement. It means that the imported ontologies are assumed to be free of heterogeneity problems. Additionally, the usage of the *ooMediator* as a target component implies that the result of mediation contains terms made available in the name space of the target ooMediator itself. For example we can have:

Listing 5. Example of a syntactic ooMediator

```

ooMediator _"http://example.org/ooMediators/firstMediatorExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  hasSource _"http://example.org/ontologies/firstSourceOntology"
  hasTarget _"http://example.org/ooMediators/firstMediatorExample"
  ...

```

- This is the case of syntactical mediators which have the role of converting an ontology (e.g. *firstSourceOntology*) from one representation language into another. Such a mediator can appear as a source component for another ooMediator. Another interesting usage would be the one in Listing 6.

Listing 6. Example of ooMediator as target component for another ooMediator

```

ooMediator _"http://example.org/ooMediators/firstMediatorExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  hasSource _"http://example.org/ontologies/firstSourceOntology"
  hasTarget _"http://example.org/ooMediators/secondMediatorExample"
  ...

ooMediator _"http://example.org/ooMediators/secondMediatorExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  hasSource _"http://example.org/ooMediators/secondMediatorExample"
  hasTarget _"http://example.org/ontologies/secondTargetOntology"
  ...

webService _"http://example.org/ws/wsExample"
  nonFunctionalProperties
    ...
  endNonFunctionalProperties
  usesMediator {_"http://example.org/ooMediator/firstMediatorExample",
                _"http://example.org/ooMediators/secondMediatorExample"}
  importsOntology {_"http://example.org/ontologies/anotherOntology",
                  _"http://example.org/ontologies/secondTargetOntology"}
  ...

```

- *firstMediatorExample* could be a syntactic ooMediator transforming *firstSourceOntology* from an arbitrary ontology language to WSMML. The transformed ontology is made available through the namespace of *secondMediatorExample*, which in its turn has the role of solving the heterogeneity problems between the transformed ontology and *secondTargetOntology*. Inside of *wsExample* we can refer to elements from *secondTargetOntology* which is aligned with the entities modelled in the *firstMediatorExample*. Please note that the *firstSourceOntology* and *secondMediatorExample* cannot be referred from *wsExample* as they are source components in one of the used mediators.

It is important to stress that the target component of an ooMediator doesn't necessarily indicate what WSMO element uses that mediator, but rather indicates where or how the mediation results are made available. Though, for example, if the ontology A imports another ontology S and uses an ooMediator M, where M has as

source the ontology S, and as target the ontology A, it means that one can refer in terms of ontology A to elements modelled in ontology S.

Mediation Service (*hasMediationService*)

The *hasMediationService* links the description of the ontology mediator (i.e. WSMO *ooMediator*) with a concrete solution for ontology mediation. This mechanism allows using *ooMediators* to describe pieces of functionality offered by complex services able to perform concrete mediation scenarios: instance transformation, query rewriting, etc.

There are three possibilities of connecting to the mediation services, using *hasMediationService*, by specifying a:

- **webService** – Directly links to a Web service able to offer the functionality described by the *ooMediator*.
- **goal** – Links to a goal that is to be used in the discovery process to find a Web service offering the functionality described by the *ooMediator*.
- **wwMediator** - Links to a web service using a *wwMediator*; the source *webService* of the *wwMediator* offers the functionality described by the *ooMediator*.

Listing 7 presents an example of such a mediation service together with the relevant aspects modelled in the ontology it imports. The service is able to perform instance transformation between two given ontologies, *location* ontology and *addresses* ontology and viceversa. The capability of this service specifies that it is able to transform instances of the concept *City* in the *location* ontology to instances of concept *city* in the *addresses* ontology. It is important to note that no details regarding what this transformation actually means are revealed in this capability but only that this transformation is taking place. Of course, if desired, such information could be included, for example in the *effect* of the capability; these details could even contain the whole ontology alignment specification (e.g. mappings expressed as WSML axioms).

Listing 7. Example of mediation service able to perform instance transformation

```

namespace { _"http://www.wsmo.org/ontologies/Mediator#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
  loc _"http://www.wsmo.org/ontologies/location#",
  addr _"http://example.org/ontologies/addresses#" }

webService _"http://www.wsmx.org/webservices/DataMediator"
  nfp
    dc#title hasValue "Data Mediator Web Service"
    dc#type hasValue _"http://www.wsmo.org/TR/d2/v1.2/#services"
    wsml#version hasValue "$Revision: 1.92 $"
  endnfp

importsOntology { _"http://www.wsmo.org/ontologies/location#",
  _"http://www.wsmo.org/ontologies/addresses#" }

capability

```

```

sharedVariables ?sourceInstance
precondition
  nonFunctionalProperties
    dc#description hasValue "The instances to be transformed have
                                to be instances of the specified concepts"
  endNonFunctionalProperties
  definedBy
    ?sourceInstance memberOf loc:City
  or ?sourceInstance memberOf addr:city.

assumption
  nonFunctionalProperties
    dc#description hasValue "The assumption made is that the
                                instance to be mediated is an instance
of
                                a concept defined in either the location
                                or the addresses ontology."
  endNonFunctionalProperties
  definedBy
    (?sourceInstance memberOf loc#?C and
    ?C subConceptOf true)
  or
    (?sourceInstance memberOf addr#?C and
    ?C subConceptOf true)

postcondition
  nonFunctionalProperties
    dc#description hasValue "A new instance of a concept from
                                the target ontology is created"
  endNonFunctionalProperties
  definedBy
    (?sourceInstance memberOf loc#City and
    mediated(?sourceInstance) memberOf addr#city)
  or
    (?sourceInstance memberOf addr#city and
    mediated(?sourceInstance) memberOf loc#City).

interface
  ...

```

We also do not exclude the possibility to point from the mediation service capability to external documents containing the full specification of the mappings that are going to be used in the mediation process by the service. As the default language used to represent WSMO ontologies is WSML, the default way of representing the ontology mappings would be by using WSML axioms. It is however possible to represent mappings using other ontology representations languages or even an abstract mapping language as the one presented in [Appendix A](#). It is then possible to specify the mapping language used, via the *usedMappingLanguage* non-functional property (if not specified the mappings are assumed to be expressed as WSML axioms).

2.3 Related Mediation Techniques

Data mediation level has a crucial role in all mediation aspects that are described in this document: the process mediation level relies on it and all four types of mediators introduced by WSMO [[Roman et al., 2005](#)] refer to it in order to achieve their functionality. Data mediation is a very well explored area and a multitude of approaches were developed in this direction. In this section our scope is not to

describe a particular solution or approach for such a mediator but to draw the requirements such a mediator has to meet (subsection [2.3.1](#)) in order to suits the needs of WSMO mediators. In addition we give a very short overview of what ontology mediation means as described in most of the existing approaches (subsection [2.3.2](#)).

2.3.1. Requirements

In this context, everything used for modeling various WSMO elements and the data being exchanged is semantically described by using ontologies. As a consequence, the first requirement that comes up for data mediation is that it has to make use of the ontological description for solving data heterogeneity problems. That is, the data mediation level has to resolve the existing mismatches between different conceptualizations used in describing a particular domain, in other words to perform what is called ontology mediation.

The very next requirement comes directly from [[Roman et al., 2005](#)] which mentions that the mediators have to allow for loose coupling between services, goals and ontologies. By this, we can conclude that from the three forms of ontology integrations identified in [[Ding et al., 2002](#)] (ontology merging, ontology alignment and ontology relating) only ontology alignment and ontology relating are of interest for us in this context. Ontology merging implies that a new ontology is created based on the existing ones and in all future processing only the new ontologies is to be used. Obviously this contradicts the above presented requirement. On the other hand, ontology alignment and ontology relating bring the ontologies into a mutual agreement and show how they are related. The main difference between these two approaches is that in the first case at least one of the input ontologies is subject of changes and adaptations while in the second case the input ontologies are kept intact and additional axioms are required to describe the relations between them. As a consequence, another requirement we impose to the data level mediator is to be able to perform ontology integration by keeping the input ontologies intact (if it is possible) or by adjusting one or more from the input ontology. In other words, we envision two usage scenarios for data mediation: the first one is to make available the set of axioms (we will call them mapping rules from now on) relating the ontologies to be picked by the used reasoning service and the second one, to provide the result of evaluating the mapping rules on a set of input data (e.g. ontology instances, queries) directly, using its own reasoning service (e.g. instance transformation, query rewriting).

Summarizing the requirements presented in the above paragraphs, the data level mediation should offer one of the following pieces of functionality:

- instances transformation based on a set of mappings identified between the input ontologies.
- query rewriting from the terms of one ontology in terms of the target ontology. This might also include the task of merging the instances of two mediated ontologies to suppress duplicates.
- access to the mapping rules relevant for a specific scenario. Note that even if no assumption is made on the internal mappings representation, the data mediator has to provide the mapping rules in the required format (e.g. WSML axioms).

Additionally the data mediation level can take care of resolving pure syntactical transformations to/from various ontology representation languages to WSML. By this, another requirement we can derive is:

- convert the ontologies from different logical languages to WSML.

It is worth noting that not all the above requirements are mandatory - depending on what requirements are fulfilled by a particular mediation service, different WSMO mediators can be created having a particular mediation service as an underlying technological solution.

2.3.2. Used Techniques

The most common approach towards data mediation in the context of Semantic Web and Semantic Web Services is ontology mapping. As data is described in terms of ontologies, mappings are created between these ontologies and applied on data in various mediation scenarios. As described in [Mocan and Cimpian, 2005] this involves a design time process consisting of a set of semi-automatic mechanisms and interaction with the domain expert, which has as output a set of mappings between ontologies. In other works ([Doan et al., 2002], [Euzenat et al., 2004]) these mappings (also called alignments) are generated automatically but in general for these approaches the degree of accuracy cannot be guaranteed. The language used to represent the mappings is usually influenced by the language used to represent the ontologies that have to be mapped. In [Mocan and Cimpian, 2005] the abstract mapping language proposed in [de Bruijn et al., 2004b] is used (the EBNF grammar of this abstract mapping language can be found in Appendix A). As the mapping language is an abstract one, it needs to be grounded to a concrete language, to associate a formal semantics to the mappings and basically to make them usable by existing reasoners for that particular concrete language. Such grounding mechanism are provided by [Mocan and Cimpian, 2005] to Flora-2 (see <http://flora.sourceforge.net>) or by [Predoiu et al., 2004] to WSML-Flight.

The second stage of the mediation process implies the usage of the design-time created mappings or alignments in different mediation scenarios as instance transformation, query rewriting, instance unification etc. This is a completely automatic, run-time process that can be wrapped in a Web service to obtain what we call in this document the mediation service. The scope of the *ooMediators* is to describe in WSMO terms, the functionality of this service.

A special case of Ontology Mediation is consider the transformation of an ontology from one representation language to another. In this case the mappings are created at the meta-level of the two languages and applied in run-time processes to effectively transform a given ontology from one language to another. In [de Bruijn et al., 2005] a set of transformation functions are provided to map WSML-Core to OWL-DL and other way around. If wrapping the implementation of these transformation functions in a Web service, a *syntactic ooMediator* can be used to describe in WSMO terms its functionality. We call these mediator a *syntactic ooMediator* because even if the mappings at the meta-level capture the semantic relationships between the two languages, the transformations operated on the ontologies themselves are only syntactical transformations.

3 GG Mediators

This section addresses the usage, definition, and mediation techniques of GG Mediators in more detail. We first outline the functional purpose of GG Mediators and provide their definition as WSMO elements. Then, we expose Δ -relations that explicitly specify the logical relationships between source and target components of a mediator and provide the basis for a new mediation technique.

3.1 Aims and Usage

The purpose of GG Mediators is to connect Goals and provide additional information on the relationships between them in order to enable more sophisticated management of Goals. Apart from resolving possible data level mismatches, the central mediation technique is so-called **Δ -relations** that define the explicit logical relationship between Goals. In particular, a Δ -relation denotes the difference between the functionality requested in related Goals, wherefore we discuss the definition and properties below in detail.

Beneficial usage scenarios of GG Mediators are:

Goal Specification by Refinement

Consider a goal $G1$ that defines the objective `buy product`, and another goal $G2$ `buy ticket` whereby `ticket` is sub-class of `product` in the used domain ontology. Imagine that $G1$ already exists, and some user wants to define $G2$. As $G2$ is a semantic refinement of $G1$ (means: both goals have the same structure, but the scope of $G2$ is narrower than the one of $G1$), we can use a ggMediator $GGM_{G1, G2}$ that contains $\Delta_{G1, G2}$ for automatically deriving the specification of $G2$ as it holds that $G2 = G1 \wedge \Delta_{G1, G2}$.

This follows the concept of problem specification by refinement which is the main purpose and motivation for Problem-Solving Methods [Fensel, 2000] for goal creation and creation of goal ontologies as collections of explicitly interlinked goals.

Goal Adjustment by Strengthening and Weakening

The second usage scenario refers to if a client specifies a goal that can not be resolved by any existing Web service, but similar goals that can be resolved. Imagine in the example setting that we have a Web service $WS1$ `book train tickets`, with `train ticket` being a sub-class of `ticket` in the used domain ontology. Imagine a Goal $G3$ `buy a flight ticket from Innsbruck to Vienna` that is a semantic refinement of the above $G2$ `buy ticket`. While $WS1$ is not usable for solving $G3$, it is applicable for $G2$. Assuming that $WS1$ allows purchasing train tickets from Innsbruck to Vienna, we can propose the client to weaken the goal $G3$ towards a Goal $G2'$ `buy a ticket from Innsbruck to Vienna` and define this in $\Delta_{G3, G2'}$ that states `all tickets from Innsbruck to Vienna that are not flight tickets`. Knowing that $WS1$ is usable for $G2$ and that $G2'$ is a proper refinement of $G2$, we can determine that $WS1$ is usable for resolving $G2'$. It is to remark that $G2'$ defines a different objective than $G3$, and hence needs to be approved by the client for resolution.

This usage scenario complies with the concept of weakening and strengthening that is realized by *Refiners* as top-level elements in the UMPL framework for describing Problem Solving Methods [Fensel et al. 2003] that allows to make resources applicable for a broader range of problems.

Goal Ontologies for Efficient Management of WSMO Elements

A third functional purpose of usage Δ -relations in GG Mediators is defining goal-ontologies as collections of goals semantically connected via GG Mediators. Imagine that from previous runs of a Web Service discovery engine, we have determined a set of Web Services that are applicable for resolving goal $G1$: $WS4G1 = \{WS1, WS2, \dots, WS_n\}$. Because $G2$ is a proper semantic refinement of $G1$, we know that the set of applicable Web Service for resolving $G2$ can only be equal or a subset of those applicable for $G1$, i.e: $WS4G2 \subseteq WS4G1$. By having defined $\Delta_{G1, G2}$ in a ggMediator with $G1$ as the source and $G2$ as the target, we can omit invocation of a discoverer for determining $WS4G2$, as it holds that those Web Services are applicable for resolving goal $G2$ that are in $WS4G1$ and satisfy $\Delta_{G1, G2}: WS \in WS4G2 \leftarrow WS \in WS4G1 \wedge match(WS, \Delta_{G1, G2})$. Assuming that most probably the invocation of a discoverer is more expensive than this rule, we can utilize Δ -relations in mediators to reduce the number of expensive operations in order to make Semantic Web Service technologies for discovery, selection, and composition more effective. Thereby, Δ -relations provide additional knowledge that can be used for minimizing the number of elements that need to be inspected in expensive operations. Following the approach of additional constraints for gaining efficiency on the reasoning process for automated problem solving as presented in [Fensel and Straatman, 1998], this allows efficient management of WSMO elements as further described in [Stollberg et al., 2005].

3.2 GG Mediator Definition

With respect to the above examinations, the following listing provides the complete definition of GG Mediators with further explanation of the description elements below.

Listing 8. GG Mediators definition

```

Class ggMediator sub-Class mediator
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  usesMediator type ooMediator
  hasSource type {goal, ggMediator}
  hasTarget type {goal, ggMediator}
  hasDeltaRelation type axiom
  hasMediationService type {goal, service, mediator}

```

Non-Functional Properties

the WSMO non-functional properties relevant for GG Mediators are the same as for all WSMO Mediator types, namely in alphabetical order (see [Roman et al., 2005] for definition): 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

denotes the ontologies used for describing the mediator, as long as no ontological mismatches need to be resolved

Using Mediator

denotes OO Mediators that are used to resolve data level mismatches between the source and target Goals

Source

denotes the Goals that are taken as a sources for defining the logical relationship between the connected Goals

Target

denotes the Goal that is created or connected on via the logical relationship to the source Goals

Delta-Relation

explicitly defines the logical relation between the source and target Goals

Mediation Service

points to a mediation service that executes the mediation definitions. While specific services are used for data and process level mediation that are capable of executing the respective mediation definitions, the mediation service for the Δ -relations is most presumably the reasoning facility applied in engines for client-side goal definition or in engines for Web Service discovery and composition.

While we discuss the definition of usage of Δ -relations as a mediation technique below, the following illustrates modeling of GG Mediators and Δ -relations between the source and target goals.

Let's consider the following goals: $G0$ for buying a product (i.e. receiving a contract of purchase for some product), and $G1$ for buying a ticket between two locations whereby ticket is a special type of product and thus defined as a sub-concept in the respective domain ontology. Obviously, $G0$ and $G1$ are correlated to each other. Hence, we can define a GG Mediator that link the goals and explicitly denote the logical relationship between them in a Δ -relation.

The following listings provide the element listings for the example, i.e. the goals and GG Mediators in WSML [de Bruijn et al., 2005]. For simplicity reasons, we assume that all elements of the example use the same ontology; in case they use different ontologies, potentially occurring data level heterogeneities are handled by usage of appropriate OO Mediators in the GG Mediators. Listing 9 gives the ontology used as the terminology definition in this use case (this is not a well-engineered ontology, but a simplification that satisfies the needs for academic showcasing).

Listing 9. Ontology Example

```

namespace{ _"http://www.wsmo.org/ontologies/ggOntologyExample#",
  dc _"http://purl.org/dc/elements/1.1#" }
ontology _"http://www.wsmo.org/ontologies/ggOntologyExample"
nonFunctionalProperties
  dc#description hasValue "ontology snippet for exemplification"
endNonFunctionalProperties
concept person
  nonFunctionalProperties
    dc#description hasValue "concept of a person"
  endNonFunctionalProperties
  name ofType _string

```

```

    requestId ofType _integer
concept city
  nonFunctionalProperties
    dc:description hasValue "concept of a city"
  endNonFunctionalProperties
  name ofType _string
  code ofType _integer
concept route
  nonFunctionalProperties
    dc:description hasValue "route between two locations"
  endNonFunctionalProperties
  startLocation ofType city
  destinationLocation ofType city
  requestId ofType _integer
concept product
  nonFunctionalProperties
    dc:description hasValue "general product description"
  endNonFunctionalProperties
  name ofType _string
  price ofType _integer
concept ticket subConceptOf product
  nonFunctionalProperties
    dc:description hasValue "tickets for traveling as a subclass of product"
  endNonFunctionalProperties
  passenger ofType person
  forRoute ofType route
concept hotel subConceptOf product
  nonFunctionalProperties
    dc:description hasValue "available hotel in a certain city"
  endNonFunctionalProperties
  name ofType _string
  inCity ofType city
  noNights ofType _integer
  requestId ofType _integer
concept car subConceptOf product
  nonFunctionalProperties
    dc:description hasValue "car to rent in a certain city"
  endNonFunctionalProperties
  inCity ofType city
  noDays ofType _integer
  requestId ofType _integer
concept contract
  nonFunctionalProperties
    dc:description hasValue "contract between two parties used for purchasing or renting"
  endNonFunctionalProperties
  buyer ofType person
  seller ofType _iri
  product ofType product

```

The following listings provide the specification of the goals in the example as described above in natural language.

```

Listing 10. G0: buy a product (receiving a contract of
           purchase for some product)

```

```

namespace {_"http://www.wsmo.org/ontologies/G0#",
  wgOnt _"http://www.wsmo.org/ontologies/ggOntologyExample#"}

```

```

goal _"http://www.wsmo.org/ontologies/G0" nonFunctionalProperties

```

```

    dc#description hasValue "goal of buying a product"
endNonFunctionalProperties

importsOntology _"http://www.wsmo.org/ontologies/ggOntologyExample.wsm!"

capability
  sharedVariables {?client, ?product}
  precondition
    nonFunctionalProperties
      dc#description hasValue "information about the buyer and the product to be purchased
are given"
    endNonFunctionalProperties
    definedBy
      ?client memberOf ggOnt#person and
      ?product memberOf ggOnt#product.
  postcondition
    nonFunctionalProperties
      dc#description hasValue "a contract for the product is generated"
    endNonFunctionalProperties
    definedBy
      ?contract [buyer hasValue ?client,
        seller hasValue ?seller,
        product hasValue ?product]
      memberOf ggOnt#contract .

interface
  choreography _"http://www.wsmo.org/ontologies/G1Choreography"

```

Listing 11. G1: buy a ticket between two locations

```

namespace {_"http://www.wsmo.org/ontologies/G1#",
  wgOnt _"http://www.wsmo.org/ontologies/ggOntologyExample#"}

goal _"http://www.wsmo.org/ontologies/G1" nonFunctionalProperties
  dc#description hasValue "goal of buying a ticket"
endNonFunctionalProperties

importsOntology _"http://www.wsmo.org/ontologies/ggOntologyExample.wsm!"

capability
  sharedVariables {?client, ?route}
  precondition
    nonFunctionalProperties
      dc#description hasValue "information about the client and the route are given"
    endNonFunctionalProperties
    definedBy
      ?client memberOf ggOnt#person and
      ?route memberOf ggOnt#route .
  postcondition
    nonFunctionalProperties
      dc#description hasValue "a contract for the ticket is to be provided"
    endNonFunctionalProperties
    definedBy
      ?ticket memberOf ggOnt#ticket and
      ?ticket[forRoute hasValue ?route] and
      ?ticketContract memberOf ggOnt#contract and
      ?ticketContract[buyer hasValue ?client,
        seller hasValue ?seller,
        product hasValue ?ticket] .

interface

```

choreography _"http://www.wsmo.org/ontologies/G1Choreography"

The GG Mediator $ggM_{G0,G1}$ defined below connects $G0$ and $G1$, including a Δ -relation that denotes the logical difference between them. The difference between $G0$ and $G1$ is that the former specifies any kind of product while the latter requests a ticket as a special kind of product, so that intuitively the Δ -relation denotes "all products that are not tickets". The next section explain the definition, computation, and usage of Δ -relations in detail.

Listing 12. ggMediator between G0 and G1

```

namespace { _"http://www.wsmo.org/ontologies/ggm1#",
  ggOnt _"http://www.wsmo.org/ontologies/ggOntologyExample#"}
ggMediator _"http://www.wsmo.org/ontologies/ggm1"
  nonFunctionalProperties
    dc#description hasValue "GG mediator between G1 and G2"
  endNonFunctionalProperties

  importsOntology _"http://www.wsmo.org/ontologies/ggOntologyExample.wsmi"

  source _"http://www.wsmo.org/ontologies/G1"
  target _"http://www.wsmo.org/ontologies/G2"

   $\Delta$ Relation
    nonFunctionalProperties
      dc#type hasValue "refinement"
    endNonFunctionalProperties
    definedBy
      forall ?x(?x memberOf ggOnt#product and neg(?x memberOf ggOnt#ticket)).

```

3.3 Related Mediation Techniques

The mediation techniques required for GG Mediators are *data level mediation* for resolving terminological mismatches between the source and target Goals, and *Δ relation mediation* for defining and handling logical relations between the source and source and target Goals explicitly. While OO Mediators as described in [Section 2](#) are used for the former, the latter is a new type of mediation technique that we expose in the following in more detail

3.3.1 Requirements

WSMO defines a Goal to represent the objective that some client wants to achieve by using Web services. Currently, WSMO Goal descriptions consist of a *requestedCapability* that specifies the functionality a client expects from a Web service in order to solve the objective, and a *requestedInterface* that is intended to define how the client wants to interact with a Web service [[Roman et al., 2005](#)]. The former can also be understood as the client's objective specification, while the latter defines the possible behavior of a Goal for consuming a Web service via its choreography interface. In order to attain additional information on the relationship of Goals with respect to the client's objective, the primary aspect of interest is the relationship between the *requestedCapability* descriptions of Goals. The relationship between *requestedInterface* descriptions of Goals is omitted at this point in time (also

with respect to that this notion possibly will be refined in future versions of WSMO Goal definitions).

The relationship between client's objective specification of Goals that we are interested in coincides with the logical relationship between the *requestedCapability* descriptions of Goals. For clarification let's consider the following example. Consider two Goals: G1 defines "buy product", and the target Goal G2 defines "buy ticket", whereby "ticket" is sub-concept of "product" in the used domain ontology. The *requestedCapability* descriptions of G1 and G2 have the same structure; the difference is that G2 restricts the object to be purchased to tickets as a subset of products. We can define this relationship as the explicit logical difference between the capability descriptions of G1 and G2. This is what we refer to as a **Δ -relation** whose definition and properties we discuss below in detail. A GG Mediator then connects G1 and G2 including the Δ -relation between them, so that we obtain an element that precisely denotes the logical relationship between goals.

The mediation technique of Δ -relations can also be used beneficially within WG and WW Mediators. In the former, Δ -relations can be defined for explicitly defining the logical differences between capability descriptions of Goals and Web Services, and in the latter as the difference between capabilities of Web Services. In case that the same elements are connected by GG, WG, and WW Mediators, certain relationships hold between the Δ -relations in the mediators. Referring to the above example, consider a Web service WS1 for purchasing train tickets as a sub-class of 'tickets' in the used domain ontology. Given G1 and WS1, the Δ -relation in a respective WG Mediator specifies "all products that are not train tickets" as the explicit logical difference between G1 and WS1. Having also given the Δ -relation between G1 and G2 as "all products that are not tickets", we can determine the Δ -relation between G2 and WS1 on basis of the known Δ -relations between G1, G2, and G1, WS1 - we discuss this in more detail below.

It is to remark that mediation by Δ -relations is different from data and process level mediation. While the latter are concerned with techniques for establishing interoperability if this is not given a priori by resolving mismatches, mediation with Δ -relations is concerned with improving the efficiency of Semantic Web service technologies. The elements that are connected via mediators in an goal or element ontology can reside in a functional manner without the additional teleological information. However, efficiency of core technologies for handling Semantic Web services is a crucial issue with respect to large-scale, industrial strength applicability. As mediation with Δ -relations can significantly improve efficiency, we consider this to be a beneficial mediation technique for Semantic Web services.

3.3.2 Used Techniques

As additional information for enabling efficient resource management, a Δ -relation specifies the explicit logical relationship between the functional descriptions of correlated resources. By functional descriptions we refer to black box descriptions of the functionality provided by a Web service or the one requested by a service requests, i.e. capabilities of Web Services or Goals in WSMO. In order to attain the desired information for beneficial resource management techniques, we understand a Δ -relation to describe additional information for establishing logical equivalence

between functional resource descriptions that have some commonality but are not equivalent.

The formal semantics for WSMO capabilities defined in [Lausen, 2005] defines existence of a commonality between two capability descriptions $C_1 = (\Phi^{pre}, \Phi^{ass}, \Phi^{post}, \Phi^{eff}, SV(C_1))$ and $C_2 = (\Psi^{pre}, \Psi^{ass}, \Psi^{post}, \Psi^{eff}, SV(C_2))$ as **capability entailment**: if there for some Web service W that is a model of C_1 in the Abstract State Space A (meaning that its execution complies with the capability description C_1) holds that W is also a model for C_2 then C_1 is considered to logically entail C_2 , formally: $C_1 \models_A C_2 \iff (W \models_A C_1 \implies W \models_A C_2)$. This capability entailment is tested for functional discovery by semantic matchmaking, wherefore the commonly defined matchmaking degrees (subsumption, plugin, intersection) denote more fine-grained logical relationships between the capability of the requester's goal and the Web service functionality, see [Keller et al., 2004], [Li and Horrocks, 2003]. Now, a Δ -relation defines additional information that allows determining logical equivalence between logical entailed capabilities. This means that given two capabilities C_1 and C_2 wherefore some capability entailment exists, Δ_{C_1, C_2} denotes additional information that need to be added to C_1 in order to obtain C_2 or vice versa.

$$C_1 \models_A C_2, \Delta_{C_1, C_2} \models C_1 \wedge \Delta_{C_1, C_2} \iff C_2$$

In order to provide all information needed for beneficial techniques for efficient resource management by Δ -relations, their description consists of two parts. The **Δ -expression** states the logical difference between functional descriptions, and the **Δ -situation** denotes the type of the relationship between them. Both can be computed for given functional descriptions, as discussed below. While the former is described by logical expressions, the latter is denoted by reserved keywords in the non-functional property *type* of a Δ -relation definition

```

 $\Delta$ Relation
  nonFunctionalProperties
    dc#type hasValue equal | plugin | subsume | intersection
  |disjoint
  endNonFunctionalProperties
  definedBy
    logical expression.

```

Following [Benjamins et al., 1996], the information desired within a Δ -expression can most appropriately be described as the logical difference between the capability descriptions. For some first-order logical expressions Φ, Ψ this is defined by $\Delta = (\Phi \wedge \neg\Psi) \vee (\neg\Phi \wedge \Psi)$. The formula for Δ is valid for arbitrary first-order logic formulas under universal closure. The below figure illustrates this definition. The universe U denotes all possible models of the information space, i.e. all possible instances of the ontology definitions used as terminology in the definition of Φ and Ψ ; the logical expressions Φ and Ψ restrict U . While $\Phi \wedge \Psi$ denotes those models common to Φ and Ψ , Δ as the logical difference is defined as disjunction of those models which are either models of Φ or models of Ψ but not common to them. For illustration, refer to the introductory example for G1 "buy product" and G2 "buy ticket" with 'ticket' being a sub-concept of 'product'. Intuitively the desired difference between G1 and G2 is "all products that are not tickets". We can write this down in first-order logic as follows (using the following symbols in F-Logic style: $:$ denotes class membership, $x[y \rightarrow z]$ denotes that x has an attribute y with the value z): $G1 = x:product$, and $G2 =$

$x:\text{ticket}$; in addition, we know from the ontology: $x:\text{ticket} \Rightarrow x:\text{product}$. So we write $\Delta_{G1, G2} = (x:\text{product} \wedge \neg(x:\text{product} \wedge x:\text{ticket})) \vee ((x:\text{product} \wedge x:\text{ticket}) \wedge \neg(x:\text{product}))$. When simplifying this following conventional tautologies in first-order logic, we obtain: $\Delta_{G1, G2} = ((x:\text{product} \wedge \neg(x:\text{product})) \vee (x:\text{product} \wedge \neg(x:\text{ticket}))) \vee (x:\text{product} \wedge \neg(x:\text{ticket}) \wedge \neg(x:\text{product})) = (x:\text{product} \wedge \neg(x:\text{ticket})) \vee \text{false} = x:\text{product} \wedge \neg(x:\text{ticket})$. This coincides with what we admired intuitively as the difference between G1 and G2.

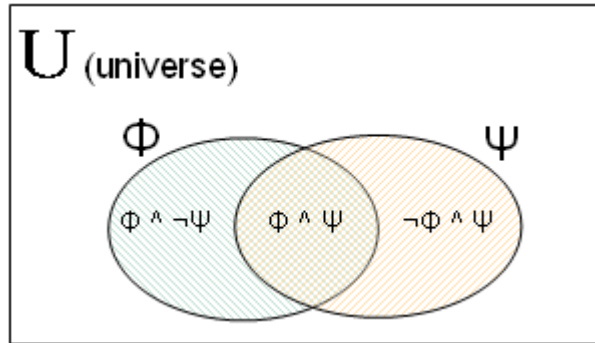


Figure 3: Definition of Δ -Relations

The Δ -expression desired for WSMO Mediators needs to precisely define the difference between the capabilities of the mediator's source and target elements. WSMO capability definitions are comprised of shared variables, preconditions, assumptions, postconditions, and effects. While the four latter elements are defined by axioms, the scope of shared variables is the complete capability that allows specifying the coherence between four other description elements. Informally, the semantics of a capability description is that if a valid pre-state is given - defined by the preconditions and assumptions - then a post-state that is defined by the postcondition and effects will be reached by executing the Web service in case that no technical execution errors occur.

A formal definition of this is given in Section 4 of [Lausen, 2005]. The interesting aspects for the definition of Δ -relations as the explicit logical difference between capabilities are (1) that models of capabilities are sets of state transitions, and (2) that the models of the capability description elements for pre- and post-states are sets of states. The latter, i.e. the preconditions, assumptions, postconditions, and effects of a WSMO capability description are defined as axioms in WSML, and hence can be interpreted as FOL formulae (at least the subset of expressions in all WSML variants that are FOL compliant - non-monotonic extensions towards Logic Programming are considered as a future object of consideration). Thus, we define the difference between two WSMO capability descriptions $C_1 = (\Phi^{\text{pre}}, \Phi^{\text{ass}}, \Phi^{\text{post}}, \Phi^{\text{eff}}, \text{SV}(C_1))$ and $C_2 = (\Psi^{\text{pre}}, \Psi^{\text{ass}}, \Psi^{\text{post}}, \Psi^{\text{eff}}, \text{SV}(C_2))$ as a tuple of the Δ -relations between the corresponding description elements: $\Delta_{C1, C2} = (\Delta_{(\Phi^{\text{pre}}, \Psi^{\text{pre}})}, \Delta_{(\Phi^{\text{ass}}, \Psi^{\text{ass}})}, \Delta_{(\Phi^{\text{post}}, \Psi^{\text{post}})}, \Delta_{(\Phi^{\text{eff}}, \Psi^{\text{eff}})})$ whereby each Δ is defined by the formula $(\Phi \wedge \neg\Psi) \vee (\neg\Phi \wedge \Psi)$ discussed above. Computing Δ -expressions with this formula most presumably results in logical expressions of higher complexity than the source expressions. In order to enable efficient reasoning on these, rewriting techniques on basis of FOL tautologies can be applied.

[Lausen, 2005] considers easier notions for capabilities like the 'implication view' ($c = \Phi^{\text{pre}} \wedge \Phi^{\text{ass}} \Rightarrow \Phi^{\text{post}} \wedge \Phi^{\text{eff}}$) will be considered in accordance to future work. The

benefit of defining complete capability as one first-order logic formula is that we could obtain Δ_{C_1, C_2} between two capabilities $C_1 = \Phi = \Phi^{pre} \wedge \Phi^{ass} \Rightarrow \Phi^{post} \wedge \Phi^{eff}$ and $C_2 = \Psi = \Psi^{pre} \wedge \Psi^{ass} \Rightarrow \Psi^{post} \wedge \Psi^{eff}$ as one first-order logic formula via $(\Phi \wedge \neg\Psi) \vee (\neg\Phi \wedge \Psi)$. However, this is an open issue as an FOL implication can only display the semantics of WSMO capability descriptions correctly when applying signature rewriting strategies in order to capture the relations between pre- and post-state descriptions.

It turns out that for most of the reasoning tasks on Δ -relations that result in functional benefits we only need to know the ontological situation that holds between the connected elements. Therefore, we distinguish five situations that can occur between two FOL formulae Φ and Ψ with respect to their models in the Abstract State Space A . Hence, we define five Δ -situations that naturally comply with the degrees of matching as identified in [Keller et al., 2004], [Li and Horrocks, 2003]. While in these are used in discovery to denote the type of commonality between logical expressions, we use them to denote the type of difference. The following defines the Δ -situations along with the simplified computation of the corresponding Δ -expression.

1. **equal**: if $\Phi = \Psi$ then $\Delta = \text{false}$. This means that the models for Φ in A are exactly the same as those for Ψ . Hence, there does not exist any logical difference between, so that Δ is empty (false).
2. **subsume**: if $(\neg\Phi \wedge \Psi) = \text{false}$, then $\Delta = (\Phi \wedge \neg\Psi)$. This means that all models for Ψ are also models for Φ but not vice versa, so that Φ logically entails Ψ : $\Phi \models \Psi$. As there can not exist any model for Ψ that is not a model for Φ so that always $(\neg\Phi \wedge \Psi) = \text{false}$, we can simplify the computation of $\Delta = (\Phi \wedge \neg\Psi)$. We say that Φ subsumes or entails Ψ .
3. **plugin**: if $(\Phi \wedge \neg\Psi) = \text{false}$, then $\Delta = (\neg\Phi \wedge \Psi)$. This is the opposite of the subsume situation, and we can the computation of Δ accordingly. We say that Φ is a plugin of Ψ , or that Ψ subsumes or entails Φ .
(the differentiation of the situations "subsume" and "plugin" gets important for enabling efficient reasoning mechanisms, see [Stollberg et al., 2005]).
4. **intersecting**: if $(\Phi \wedge \neg\Psi) \neq \text{false}$ and $(\neg\Phi \wedge \Psi) \neq \text{false}$, then $\Delta = (\Phi \wedge \neg\Psi) \vee (\neg\Phi \wedge \Psi)$. This means that there are models for Φ that are not models of Ψ and vice versa. We can not simplify the computation of Δ in this situation.
5. **disjoint**: if $(\Phi \wedge \Psi) = \text{false}$ then $\Delta = \text{false}$. This means that there does not exist any model for Φ in A that also is a model for Ψ in A so that there is no commonality between the two. Although formally the logical difference is infinitely big, we consider it to be false as there is no sensible usage of Δ -relations when Φ and Ψ are disjoint.

4 WG Mediators

A WSMO wgMediator explicitly states the relation between a Web service and a goal and resolves the possible mismatches between them. These mismatches may appear between the requested and the provided capabilities, or as well between the requested and provided choreographies. Considering the levels of mediation described in the beginning of this deliverable, the capabilities mismatches can be addressed using data mediation techniques (by means of an ooMediator), while the

choreographies mismatches can be address only by combining both data and process level mediation technologies (using an ooMediator for data heterogeneity, a ggMediator for expressing the logical relationships, and process mediation for solving the communication mismatches).

In this section we first describe the usage of wgMediators, continuing with their definition as WSMO elements and the possible mediation techniques related to wgMediators.

4.1. Aims and Usage

Depending on which are the source and target of the mediator the wgMediator may serve two different purposes:

1. Link to a Web service via its choreography interface meaning that the Web service (totally or partially) fulfills the goal which links to it;
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 fulfill the functionality described in its capability.

For this, the wgMediators have the following functionalities:

- Accomodate the possible communication mismatches. This aspect of mediation is needed during the actual invocation of a service.
- Specify relation between the Web service and the goal (the Web service may partially or totally fulfil the goal). This aspect of mediation may be needed both for discovering a Web service able to fulfil a certain goal and for linking a Web service (via its orchestration) to a goal. Considering a Web service WS and a goal G, the relation between them can be specified either direct, by defining the corresponding Δ relation (as defined in the previous chapter) or indirect, by identifying a goal G_0 that perfectly match WS and for which the Δ relation between it and G was already defined.
- For runtime data mediation, by means of an OOMediator.

4.2. WG Mediator Definition

The following listing provides the wgMediator's definition

Listing 13. WG Mediators definition

```

Class wgMediator sub-Class mediator
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  hasSource type {webService, goal, wgMediator, ggMediator}
  hasTarget type {webService, goal, ggMediator, wgMediator}
  usesMediator type {ooMediator, ggMediator}
  hasDeltaRelationDefinition type axiom
  hasMediationService type {webService}

```

Non-Functional Properties

the wgMediator has the same non-functional properties as the mediator: 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 [[Roman et al., 2005](#)]

Importing Ontology

used to import ontologies needed by the mediator, as long as no conflicts need to be resolved

Source

the source components defines the entity that is the source of the mediator. For the wgMediator the source component can be a goal, a Web Service, a wgMediator or a ggMediator

Target

the target component defines the entity that is the target of the mediator. For the wgMediator the source component can be a goal, a Web Service, a wgMediator or a ggMediator

Using Mediator

a wgMediator may use a set of ooMediators in order to map between different vocabularies used in the description of goals and Web services, and a set of ggMediators in order to simplify the definition of Δ Relation

Δ Relation Definition

denotes the explicit definition of the ontological relationship between the source and the target. The wgMediator defines the relations between the two capabilities; these relations are used during the discovery process

Mediation Service

points to a mediation service that executes the mediation. The mediation service is used during the actual invocation of a Web Service, for accommodating the mismatches between the choreographies of the two participants in the conversation.

4.3. Related Mediation Techniques

The mediation techniques for WG Mediators are first data mediation by usage of OO Mediators, Δ relation mediation for expressing the logical relationship between the source and target elements, and communication mediation for resolving mismatches between the Choreography Interface definitions of the source and target components.

The data level mediation and the Δ relation mediation are described in the previous chapters of this deliverable, and they will not be further addressed in this section. This section describes the communication level mediation between the two participants.

4.3.1. Requirements

We consider the communication mediation as being that level of mediation which is solving the heterogeneity problem from the communication point of view. A Communication Mediator should be able to analyze the public processes of the two communication partners (as declared in their interfaces), and to determine how these two partners can actually communicate. Since it is dealing with public processes heterogeneity, the communication mediator is also called process mediator in [WSMX](#).

[Fensel and Bussler, 2002] identifies three possible cases that may appear during the message exchange:

- Precise match. The two partners have exactly the same pattern in realizing the business process, which means that each of them sends the messages in exactly the order the other one requests them. In this ideal case the mediator is not needed, the communication taking place without any problem.
- Resolvable message mismatch. This case appears when the two partners use different exchange patterns, but several transformations can be performed in order to resolve the mismatches (for example when one partner sends more than one concept in a single message, but the other one expects them separately. In this case the mediator can “break” the initial message, and send the concepts one by one).
- Unresolvable message mismatch. In this case, one of the partners expects a message that the other one do not intend to send (for example, an acknowledgement). Unless the mediator can provide this message, the communication reaches a dead-end (one of the partners is waiting indefinitely).

[Cimpian and Mocan, 2005] propose an architecture and an algorithm for a Process Mediation prototype able to solve a number of resolvable message mismatches:

- Stopping an unexpected message - If one of the partners sends a message that the other one does not want to receive, the mediator should just retain and store it. This message can be sent later, if needed, or it can just be deleted after the communication ends.
- Inversing the order of messages - If one of the partners sends the messages in a different order than the other partner expects, the messages that are not yet expected will be stored and sent when needed.
- Splitting a message - If one of the partners sends in a single message multiple information that the other one expects to receive in different messages, the information can be split and sent in a sequence of separate messages.
- Combining messages - If one of the partners expects a single message, containing information sent by the other one in multiple messages, the information can be combined into a single message.
- Sending a dummy acknowledgement - If one of the partners expects an acknowledgement for a certain message, and the other partner does not intend to send it, even if it receives the message, an acknowledgment can be automatically generated and sent to the partner which requires it.

4.3.2. Used Techniques

Since the architecture and implementation of this prototype are out of the scope of this document, no details are presented here, but only an example for illustrating the types of mismatches addressed by this prototype.

The runtime process mediation is based on the following assumption: since the goal contains the requested interface, which is the interface of the service it wants to invoke, a different component (or a subcomponent of the process mediator) will have to inverse this requestedInterface in order to obtain the actual interface of the goal. That is, if in the choreography requested by the goal, the `out` list contains the

concept *x*, this actually means that the service will have to send to the goal instances of this concept. In the goal's choreography that will be sent to the process mediator, concept *x* will be part of the *in* list, and not of the *out* list. This is needed in order to treat both the requestor and the provider of the service as equal partners, disregarding the fact that one of them is the client and the other one the service.

Additionally, we assume that the two choreographies are correctly modelled, the process mediation not being able to compensate for the incorrect or insufficient information (for example the goal's choreography can not state that a name, that is a *string*, has to be sent; instead, the *person* or the *city* with that certain name will be sent).

In order to illustrate the functionality of the process mediator, we will define a part of the choreography of a Web service offering travel tickets between two locations and accomodation in the destination city, and a part of the inversed choreography of G3 (the description of the entire choreographies of both the requestor and the provider would complicate too much the example).

Listing 14. Choreography of G3

```

interface G3Interface
  choreography G3Choreography
  stateSignature
    importsOntology _"http://www.wsmo.org/ontologies/ggOntologyExample"
  in
    ggOnt#contract withGrounding _"..."
  out
    ggOnt#person withGrounding _"...",
    ggOnt#route withGrounding _"..."
  concept completed
  nonFunctionalProperties
    dc#description hasValue "a certain request has been completed"
  endNonFunctionalProperties
  requestId ofType _integer

  transitionRules
    /*
    * a route is created and sent only after a person has a request
    */
    forAll ?person with (?person[
      requestId hasValue ?requestId
    ] memberOf ggOnt#person
      add(?route[
        requestId hasValue ?requestId
      ] memberOf ggOnt#route)
    endForAll

    /*
    * expect contract after route exists
    */
    forAll ?contract with (?contract[
      product hasValue ?route and
      buyer hasValue ?person
    ] memberOf ggOnt#contract
      add(?completed[
        requestId hasValue ?requestId
      ] memberOf completed)
    endForall

```

Listing 15. Choreography of the Web service

```

interface Tickets&HotelsInterface
  choreography Tickets&HotelsChoreography
    stateSignature
    importsOntology _ "http://www.wsmo.org/ontologies/ggOntologyExample"
    concept confirmation
    nonFunctionalProperties
      dc#description hasValue "confirmation for the existence of a route"
    endNonFunctionalProperties
    forRoute ofType ggOnt#route
  in
    ggOnt#route withGrounding _ "...",
    ggOnt#person withGrounding _ "...",
  out
    ggOnt#contract withGrounding _ "...",
    confirmation withGrounding _ "...",

  transitionRules

  /*
  * when route is received, create an instance of confirmation
  */
  forall ?route with (?route[
    requestId hasValue ?requestId
  ] memberOf ggOnt#route
  ) add(?confirmation[
    forRoute hasValue ?route
  ] memberOf confirmation
  ) endforall

  /*
  * when person is received, having the same requestId, the contract is sent
  */
  forall ?person with (?person[
    requestId hasValue ?requestId
  ] memberOf ggOnt#person
  ) add(?contract[
    product hasValue ?route and
    buyer hasValue ?person]
    seller hasValue "http://www.wsmo.org/ontologies/Tickets&Hotels#"
  ] memberOf ggOnt#contract)
  ) endforall

```

A graphical representation of the messages exchanged between the two parties is illustrated in the following figure:

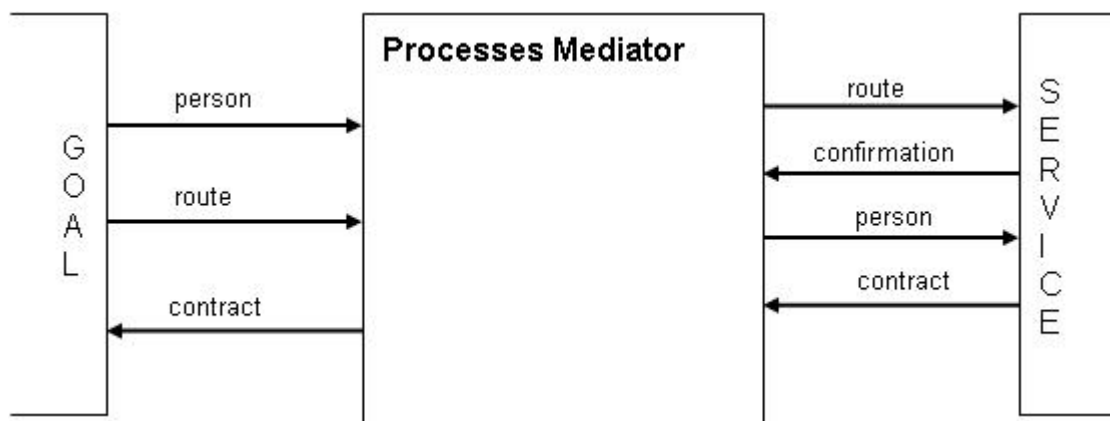


Figure 4. Web Service - G3 interaction

In this case, the Process Mediator will have to:

1. Store the instance of person for further use
2. Forward the instance of route
3. Retain and delete the confirmation (it is not expected by the goal)
4. Retrieve the instance of person from the internal repository
5. Forward the instance of person

Details about how the process mediator will performed all these can be found in WSMX deliverable D13.7 ([\[Cimpian and Mocan, 2005\]](#)).

5 WW Mediators

As the fourth and final WSMO mediator type, WW Mediators resolve heterogeneity problems between Web Services.

Since different Web Services are communicating between one another, mismatches can occur on data level, between capabilities of interacting Web services, on the communication level, or on the cooperation level. Data level mismatches are handled by using respective OO Mediators (see [Section 2.3](#)), differences between capabilities can be expressed and handled by Δ -relations as defined in [Section 3.3](#), and communication mismatches potentially arising between Web services are handled with with the techniques introduced in [Section 4.3](#). The definition of WW Mediators as well as cooperation level mediation techniques are aspects of future work.

6 Conclusions

This document presented a refinement of the Mediator, one of the four top level elemene of the Web Service Modeling Ontology: WSMO. The four types of mediators presented here ooMediator, wwMediator, wgMediator and ggMediator are used to cope the heterogeneity problems that potentially arise at different levels. We gave a definition and indication of usage for these mediators, a reference implementation may be found in the Web Service Modeling Execution environment WSMX [WSMX].

References

[Altenhofen et al., 2005] Altenhofen, M.; Börger, E.; Lemcke, J.: *An Execution Semantics for Mediation Patterns*. In Proceedings of the WIW 2005 Workshop on WSMO Implementations, Innsbruck, Austria, 2005; available at CEUR Workshop Proceedings Vol 134 (<http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-134/>).

[Barros and Borger, 2005] A. Barros and E. Börger: A Compositional Framework for Service Interaction Patterns and Interaction Flows, available at <http://sky.fit.qut.edu.au/~dumas/ServiceInteractionPatterns/InteractionPatternsFormalisation.pdf>.

[Benjamins et al., 1996] Benjamin, V. R.; Fensel, D.; Straatman, R: *Assumptions of Problem-Solving Methods and their Role in Knowledge Engineering*. In Proc. of the ECAI 1996, pp. 408-412, John Wiley and Sons, 1996.

[Börger, 1998] Egon Börger: "High Level System Design and Analysis Using Abstract State Machines", Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods, p.1-43, October 07-09, 1998

[BPT, 2003] Business process Trends, Glossary, available at http://www.bptrends.com/resources_glossary.cfm, 2003

[Cimpian and Mocan, 2005] E. Cimpian and A. Mocan: *Process Mediation in WSMX*. WSMX Working Draft v0.1. available at <http://www.wsmo.org/TR/d13/d13.7/v0.1>

[de Bruijn et al., 2004] Jos de Bruijn and Francisco Martin-Recuerda and Dimitar Manov and Marc Ehrig : *State-of-the-art Survey on Ontology Merging and Aligning V1*. Technical Report, SEKT project deliverable D4.2.1

[de Bruijn et al., 2004b] J. de Bruijn, Douglas Foxvog, Kerstin Zimmerman: *Ontology mediation patterns library v1*. Technical Report, SEKT project deliverable D4.3.1, 2004.

[de Bruijn et al., 2005] J. de Bruijn (ed.): *The Web Service Modeling Language WSMML*. WSMML Final Working Draft v0.2, 20 March 2005, available at <http://www.wsmo.org/TR/d16/d16.1/v0.2/>

[Ding et al., 2002] Ying Ding, Dieter Fensel, Michel C. A. Klein, and Borys Omelayenko. The semantic web: yet another hip? *Data Knowledge Engineering*, 41(2- 3):205–227, 2002.

[Doan et al., 2002] A. Doan, J. Madhavan, P. Domingos, A. Halevy: Learning to Map Between Ontologies on the Semantic Web. In *WWW2002*, 2002.

[Euzenat et al., 2004] J. Euzenat, D. Loup, M. Touzani, P. Valtchev: Ontology alignment with OLA. In Proc. 3rd ISWC2004 workshop on Evaluation of Ontology-based tools (EON), Hiroshima, Japan, pp59-68, 2004

[Feier et al., 2005] Feier, C.; Roman, D.; Polleres, A.; Domingue, J.; Stollberg, M. and Fensel, D.: *Towards Intelligent web Services: Web Service Modeling Ontology (WSMO)*. In Proceedings of the International Conference on Intelligent Computing (ICIC) 2005, Hefei, China, August 23-26, 2005.

[Fensel, 2000] D. Fensel: *Problem-Solving Methods: Understanding, Development, Description, and Reuse*. Lecture Notes on Artificial Intelligence, no 1791, Springer-Verlag, Berlin, 2000.

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

[Fensel et al. 2003] Fensel, D. et al.: *The Unified Problem Solving Method Development Language UPML*. In Knowledge and Information Systems Journal (KAIS) 5(1), 2003.

[Fensel and Straatman, 1998] Fensel, D. and Straatman, R.: *The Essence of Problem-Solving Methods: Making Assumptions to Gain Efficiency* International Journal of Human-Computer Studies 48(2): 181-215, 1998.

[Keller et al., 2004] U. Keller, R. Lara, A. Polleres, I. Toma, M. Kifer and D. Fensel: *WSMO Web Service Discovery*, WSMO deliverable D5.1 v0.1 available at http://www.wsmo.org/2004/d5/d5.1/v0.1/20041112/d5.1v0.1_20041112.pdf, 2005.

[Li and Horrocks, 2003] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology, 2003.

[Mocan and Cimpian, 2005] A. Mocan and E. Cimpian: *WSMX Data Mediation*. WSMX Working Draft v0.2. available at <http://www.wsmo.org/TR/d13/d13.3/v0.2/>

[Papakonstantinou et al., 1996] Papakonstantinou, Y; Garcia-Molina, H.; Ullman, J. D.: *MedMaker: A Mediation System Based on Declarative Specifications*. In Proceedings of the Twelfth International Conference on Data Engineering, 1996, pp 132 - 141.

[Predoiu et al, 2004] Livia Predoiu, Francisco Martin-Recuerda, Axel Polleres, Cristina Feier, Fabio Porto, Jos de Bruijn, Adrian Mocan and Kerstin Zimmermann. *Framework for representing ontology networks with mappings that deal with conflicting and complementary concept definitions*. Deliverable D1.5, DIP, 2004. Available at <http://dip.semanticweb.org/>.

[Kiryakov et al., 2005] A. Kiryakov, C. Drumm, L. Al-Jadir, A. Boukottaya, E. Kilgarriff, J. Quantz : D5.2 Mediation Module Specification: Business Data-Level Mediation - *Dip project deliverable*

[Kiryakov et al., 2005b] A. Kiryakov (Edt), Stoyan Atanassov, Dimitar Manov : Ontology Mapping Store : <http://www.omwg.org/tools/omapstore/v0.1/FactSheet.html>

[Kalfoglou et al., 2003] Yannis Kalfoglou and Marco Schorlemmer : Ontology Mapping: the State of the Art, Knowledge Engineering Review, volume 18, p 1-31

[Lausen, 2005] Lausen, H. (ed.): *Functional Description of Web Services*. WSMO Working Draft D28.1, 2005-10-20; most recent version available at: <http://www.wsmo.org/TR/d28/d28.1/v0.1/>.

[Roman et al., 2005] D. Roman, U. Keller, H. Lausen (eds.): *Web Service Modeling Ontology*, WSMO Final Draft, version 1.2 available at <http://www.wsmo.org/TR/d2/v1.2/>

[Scicluna et al., 2005] J. Scicluna, A. Polleres, D. Roman, C. Feier (eds.): *Ontology-based Choreography and Orchestration of WSMO Services*, WSMO deliverable D14 v0.1. available at <http://www.wsmo.org/TR/d14/v0.2/>, 2005.

[Stollberg et al, 2005] Stollberg, M.; Cimpian, E.; Fensel, D, *Mediating Capabilities with Delta-Relations*. Submitted to the First International Workshop on Mediation in Semantic Web Services, co-located with the Third International Conference on Service Oriented Computing (ICSOC 2005), Amsterdam, the Netherlands, December 2005.

[Stollberg et al., 2004] M. Stollberg, U. Keller, A. Mocan : *On WSMO Mediators - WSMO internal study*.

[Wiederhold, 1994] Wiederhold, G.: *Mediators in the architecture of the future information systems* Computer 25(3), 1994, pp. 38-49.

Appendix

Appendix A : Ontology Mapping Language

This Appendix contains the Abstract Syntax of the mapping language developed in [\[de Bruijn et al., 2004b\]](#) and it is a direct copy of the corresponding appendix in the indicated document in order to provide an insight to the reader reversing the possible mapping language that can be used by the mediation services.

The abstract syntax is written in the form of EBNF, similar to the OWL Abstract Syntax. Any element between square brackets '[' and ']' is optional. Any element between curly brackets '{' and '}' can have multiple occurrences.

Each element of an ontology on the Semantic Web, whether it is a class, attribute, instance, or relation, is identified using a URI. In the abstract syntax, a URI is denoted with the name **URIReference**. We define the following identifiers:

Listing A.1. Identifiers

```
mappingDocumentId ::= URIReference
ontologyId ::= URIReference
classId ::= URIReference
propertyId ::= URIReference
attributeId ::= URIReference
relationId ::= URIReference
individualId ::= URIReference
```

We allow concrete data values. The abstract syntax for data values is taken from the OWL abstract syntax:

Listing A.2. Literals

```
dataLiteral ::= typedLiteral | plainLiteral
typedLiteral ::= lexicalForm'^'^URIReference
plainLiteral ::= lexicalForm['@' languageTag]
```

The lexical form is a sequence of unicode characters in normal form C, as in RDF. The language tag is an XML language tag, as in RDF.

First of all, the mapping itself is declared, along with the ontologies participating in the mapping.

Listing A.3. Mapping Set

```

mapping ::= 'Mapping(' [ mappingId ]
             { 'source(' ontologyId ') ' }
             'target(' ontologyId ') '
             { directive } ') '

```

A mapping consists of a number of annotations, corresponding to non-functional properties in WSMO [Roman et al., 2005], and a number of mapping rules. The creator of the mapping is advised to include a version identifier in the non-functional properties.

Listing A.4. Annotation

```

directive ::= annotation | expression

annotation ::= 'Annotation(' propertyID URIReference ') '
              | 'Annotation(' propertyID dataLiteral ') '

```

Expressions are either class mappings, relation mappings, instance mappings or arbitrary logical expressions. The syntax for the logical expressions is not specified; it depends on the actual logical language to which the language is grounded.

A special kind of relation mappings are attribute mappings. Attributes are binary relations with a defined domain and are thus associated with a particular class. In the mapping itself the attribute can be either associated with the domain defined in the (source or target) ontology or with a subclass of this domain.

A mapping can be either uni- or bidirectional. In the case of a class mapping, this corresponds with class equivalence and class subsumption, respectively. In order to distinguish these kinds of mappings, we introduce two different keywords for class, relation and attribute mappings, namely 'unidirectional' and 'bidirectional'. Individual mappings are always bidirectional. Unidirectional and bidirectional mappings are differentiated with the use of a switch. The use of this switch is required.

It is possible, although not required, to nest attribute mappings inside class mappings. Furthermore, it is possible to write an axiom, in the form of a class condition, which defines general conditions over the mapping, possibly involving terms of both source and target ontologies. Notice that this class condition is a general precondition for the mapping and thus is applied in both directions if the class mapping is a bidirectional mapping. Notice that we allow arbitrary axioms in the form of a logical expression. The form of such a logical expression depends on the logical language being used for the mappings and is thus not further specified here.

Listing A.5. Class mapping

```

expression ::= 'classMapping('
                'unidirectional'|'bidirectional'
                { annotation }
                classExpr classExpr
                { attributeMapping }
                { classCondition }
                [ {' logicalExpression ' } ] ') '

```

There is a distinction between attributes mapping in the context of a class and attributes mapped outside the context of a particular class. Because attributes are defined locally for a specific class, we expect the attribute mappings to occur mostly inside class mappings. The keywords for the mappings are the same. However, attribute mappings outside of the context of a class mappings need to be preceded with the class identifier, followed by a dot '.'.

Listing A.6. Attribute mapping (to be used inside of a classMapping)

```
attributeMapping ::= 'attributeMapping('
    'unidirectional'|'bidirectional'
    attributeExpr attributeExpr
    { attributeCondition } ')'
```

Listing A.7. Attribute mapping (to be used outside of a classMapping)

```
expression ::= 'attributeMapping('
    'unidirectional'|'bidirectional'
    attributeExpr attributeExpr
    { attributeCondition }
    [ '{' logicalExpression '}' ] ')'
```

Listing A.8. Relation mapping

```
expression ::= 'relationMapping('
    'unidirectional'|'bidirectional'
    relationExpr relationExpr
    { relationCondition }
    [ '{' logicalExpression '}' ] ')'
```

Listing A.9. Instance mapping

```
expression ::= 'instanceMapping( individualID individualID )'
```

Listing A.10. Class attribute mapping

```
expression ::= 'classAttributeMapping('
    'unidirectional'|'bidirectional'
    classExpr attributeExpr
    [ '{' logicalExpression '}' ] ')'
```

Listing A.11. Class relation mapping

```
expression ::= 'classRelationMapping('
    'unidirectional'|'bidirectional'
    classExpr relationExpr
    [ '{' logicalExpression '}' ] ')'
```

Listing A.12. Class instance mapping

```

expression ::= 'classInstanceMapping(
                'unidirectional'|'bidirectional'
                classExpr individualId
                [ '{' logicalExpression '}' ] )'

```

Listing A.13. Logical Expression

```

expression ::= '{' logicalExpression '}'

```

For class expressions we allow basic boolean algebra. This corresponds loosely with Wiederhold's ontology algebra [37]. Wiederhold included the basic intersection and union, which correspond with our and and or operators. Wiederhold's difference operator corresponds with a conjunction of two class expressions, where one is negated, i.e. for two class expressions C and D , the different $C-D$ corresponds with $\text{and}(C, \text{not}(D))$.

The join expression is a specific kind of disjunction, namely a disjunction with an additional logical expression which contains the precondition for instances to be included in the join.

Listing A.14. Class expression

```

classExpr ::= classId
              |'and(' classExpr classExpr { classExpr } )'
              |'or(' classExpr classExpr { classExpr } )'
              |'not(' classExpr )'
              |'join(' classExpr classExpr { classExpr }
              [ '{' logicalExpression '}' ] )'

```

Attribute expressions are defined as such, allowing for inverse, transitive close, symmetric closure and reflexive closure, where $\text{inverse}(A)$ stands for the inverse of A , $\text{symmetric}(A)$ stands for the symmetric closure of A (notice that the symmetric closure of an attribute is equivalent to the union of the attribute and its inverse: $\text{or}(A, \text{inverse}(A))$), $\text{reflexive}(A)$ stands for the reflexive closure of A (the reflexive closure of an attribute includes for each value in the domain a tuple with equivalent domain and range v : $\langle v, v \rangle$) and $\text{trans}(A)$ stands for the transitive closure of A :

Listing A.15. Attribute expression

```

attributeExpr ::= attributeId
                  |'and(' attributeExpr attributeExpr { attributeExpr } )'
                  |'or(' attributeExpr attributeExpr { attributeExpr } )'
                  |'not(' attributeExpr )'
                  |'inverse(' attributeExpr )'
                  |'symmetric(' attributeExpr )'
                  |'reflexive(' attributeExpr )'
                  |'trans(' attributeExpr )'

```

Relation expressions are defined similar to class expressions:

Listing A.16. Relation expression

```

relationExpr ::= relationId
  | 'and(' relationExpr relationExpr { relationExpr } ') '
  | 'or(' relationExpr relationExpr { relationExpr } ') '
  | 'not(' relationExpr ') '

```

Listing A.17. Class conditions

```

classCondition ::= 'attributeValueCondition('
  attributeId
  ( individualID | dataLiteral )
  ') '
classCondition ::= 'attributeTypeCondition('
  attributeID
  classExpr
  ') '
classCondition ::= 'attributeOccurrenceCondition('
  attributeID
  ') '

```

Listing A.18. Attributeconditions

```

attributeCondition ::= 'valueCondition('
  ( individualID
  | dataLiteral )
  ') '
attributeCondition ::= 'typeCondition('
  classExpression
  ') '

```

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 in the projects RW² and TCP under the FIT-IT programme.

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