



# D13.3v0.2 WSMX Data Mediation

WSMX Working Draft 16 May 2005

**This version:**

<http://www.wsmo.org/TR/d13/d13.3/v0.2/20050516>

**Latest version:**

<http://www.wsmo.org/TR/d13/d13.3/v0.2/>

**Previous version:**

<http://www.wsmo.org/TR/d13/d13.3/v0.2/20050321>

**Editor:**

Adrian Mocan

**Authors:**

Adrian Mocan  
Emilia Cimpian

**Reviewer:**

Jos de Bruijn

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

---

## Table of Contents

### **1. Introduction**

- [1.1. General Considerations](#)
- [1.2. Overview of the Document](#)

### **2. Problem Definition and Scope of Ontology Mediation in WSMX**

- [2.1. Problem Definition](#)
- [2.2. Addressed Ontologies](#)
- [2.3. Scope of Ontology Mediation in WSMX](#)

### **3. Requirement Analysis**

- [3.1. Ontology Mediation in WSMX](#)
- [3.2. Ontology to Ontology Mediation Problems - Ontology Mismatches](#)

### **4. WSMX Data Mediation Approach**

### **5. Mappings**

- [5.1. Abstract Mappings](#)
- 5.2. Mappings Creation**
  - [5.2.1. General Strategies](#)
  - [5.2.2. PartOf View](#)
  - [5.2.3. InstanceOf View](#)
  - [5.2.4. RelatedBy View](#)
- [5.3. Conversion Functions](#)
- [5.4. Conversion Functions Creation](#)
- [5.5. Mapping Rules Generation](#)
- [5.6. Execution](#)

**6. Addressed Ontology Mismatches**

- 6.1. Different composition of concepts
- 6.2. Incomplete/over-complete concept representation
- 6.3. Data-type mismatch
- 6.4. Cardinality

**7. Flora-2 - A Reasoner for WSMX Data Mediator**

- 7.1. From WSMML to Flora-2
- 7.2. Expressing Mapping Rules in Flora-2

**8. Prototype Implementation**

- 8.1. Graphical User Interface
- 8.2. Mapping Rules Generator Module
- 8.3. Execution Environment

**9. Related Work**

- 9.1. PROMPT
- 9.2. MAFRA

**10. Conclusions and Further Directions****References****Acknowledgements****Appendix A. WSMML-Core to Flora-2 Translation****Appendix B. Changelog**

# 1. Introduction

## 1.1. General Considerations

This deliverable presents our approach to mediation in the Web Service Execution Environment [WSMX, 2005], an environment that is designed to allow dynamic mediation, selection and invocation of web services. WSMX has as scope the domain defined by Web Service Modeling Ontology (WSMO) [WSMO, 2004], thus providing a testbed and a proof for the ideas presented in WSMO. The WSMX work includes the establishing of a conceptual model, the description of the execution semantics [WSMX Execution Semantics, 2005], and an architectural and software design [WSMX Architecture, 2005], together with a working implementation of the system. As part of this work, this document describes the data mediation problems addressed in this context, together with the appropriate solutions and a software implementation. The functionality of the mediator system proposed in this document fits in the functionality of the ooMediator described in WSMO.

It is well known that models or ontologies describing the same or related problem domains are created by different entities around the world. This is why more and more systems and applications require mediation in order to be able to integrate and use heterogeneous data sources. Mapping between models is required in several classes of application, as *Information Integration and Semantic Web, Data Migration* or *Ontology Merging* [Madhavan et al., 2002].

Unfortunately, it is impossible for a mapping tool to automatically generate mapping rules. Some of the best results of research projects in this area are mapping tools that are able to validate or to suggest possible mappings, but at various points of the mapping process, domain expert intervention still remains a necessity. Efforts to build such tools concentrate on two main areas [Madhavan et

al., 2002]: the first is the design of heuristics based on structure, on non-functional properties and on names of the concepts involved in order to generate the most plausible mappings [Noy & Musen, 2000]. Sometimes, domain independent heuristics augmented by more specific and domain related heuristics can be used [Mitra et al., 2000]. The second approach is to use machine learning techniques to obtain the required mappings semi-automatically. For example, [Doan et al., 2002] use the instances of one ontology to learn a classifier for it, and then apply this classifier to classify the instances of the second ontology, and vice versa. In this way, the probability that two concepts are similar can be approximated using the fraction of instances that belong to both of them.

The data mediation solution presented herein follows the first approach described above: we propose well defined strategies and methodologies for the mapping process in order to guarantee as correct and complete mappings as possible, together with a set of algorithms and strategies meant to make the mapping task much easier (consisting of simple validations and choices). Machine-learning-based strategies are not suited to our needs because these methods require the usage of large sets of training data (in our case, instances) for obtaining a high accuracy of the predicted mappings. But the aim of the execution environment mediation is, for a given instance (obtained at the runtime), from the source ontology, to provide the corresponding instance (or instances) from the target ontology. Additionally, one has to keep in mind that each instance that has to be mediated contains a piece of information about internal business processes and behaviour, and almost no company or enterprise will agree to grant access to its business information (i.e. all its available instances), not even for the laudable purpose of allowing the learning of useful mappings for its future cooperations. That is, no training data-set is available, but only the instance that has to be mediated.

## 1.2. Overview of the Document

The entire process of mediation described here consists of three main steps: the creation of mappings, the creation of appropriate mapping rules and the execution of the mapping rules. The first two parts are carried out at the schema level: all the mappings are created considering the schema information, with possible references, through the conditional statements, to the instance set used in the modelling process. Furthermore, the fact that the execution environment scope is the WSMO domain implies that the mediated ontologies conform to the WSMO conceptual model for ontologies, having the same meta-level definition. The third step, the execution, acts on the instance data taking as input source instances and having as output the target, mediated, instances.

Although this deliverable describes the execution environment mediation, the first of the three steps presented above, takes place outside the execution environment. Its implementation consists of a graphical environment that allows the domain expert to place his or her inputs, and the implementation of the required processes that assure the semi-automatic mediation behaviour of the module (e.g. the similarity computing process). The implementations of the second and the third steps are included in the WSMX architecture and provide means of translating the given source instances into target instances, based on the previously created mappings.

This document is structured as follows: Section 2 presents the problem definition

and scope of ontology mediation in WSMX. Section 3 presents the requirements that have to be achieved by WSMX data mediation together with an exhaustive classification of potential ontology mediation mismatches. Sections 4, 5 and 6 introduce the terminology adopted in our approach and describe the solutions proposed for ontology mediation and the set of addressed problems. Details about the reasoning system used by WSMX Data Mediator, the relation between the formal language used to express WSMO ontologies and the mappings, and the mapping rules to be executed by the reasoner are described in Section 7. Section 8 briefly presents the prototype that implements the ideas presented in the previous sections. Finally, Section 9 offers an overview of two related works in this area and Section 10 draws some conclusions and refers to the further directions of WSMX data mediation.

## 2. Problem Definition and Scope of Ontology Mediation in WSMX

This section provides an overview of the domain problem and of the scope of the mediation. The first subsection presents the problem definition; the second subsection offers a short description of the ontologies used in our mediation process, and finally the last subsection outlines the scope of ontology mediation in WSMX.

### 2.1. Problem Definition

In recent years, one of the solutions adopted for dealing with heterogeneity over the World Wide Web has been to add semantic information to the data. This approach will lead in the near future to the development of a high number of ontologies, modeling aspects of the real world. But unfortunately a new problem has arisen: how to integrate applications that are using different conceptualizations of the same domain. Furthermore, the Web services enriched with semantics offered by ontologies aim to enable dynamic and strongly decoupled communication and cooperation between different entities.

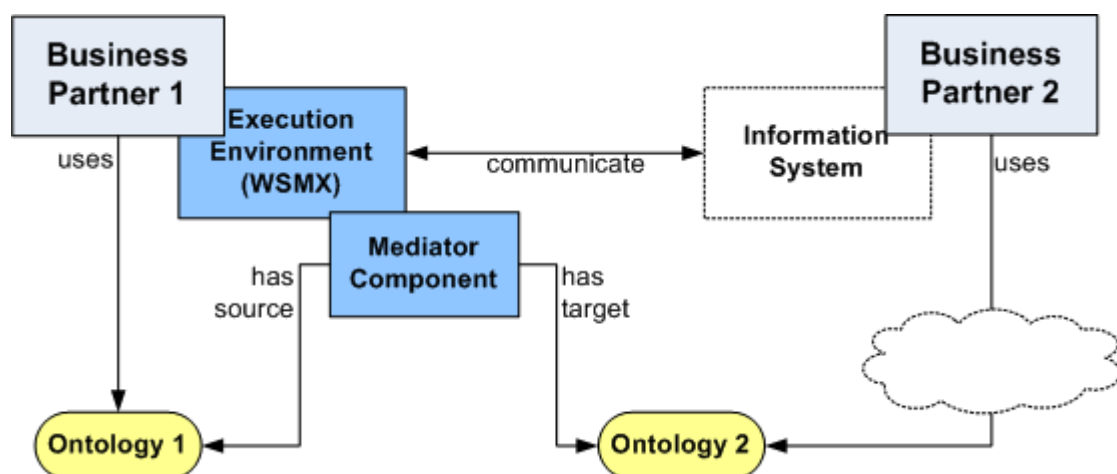


Figure 1. WSMX Data Mediation Scenario

As a consequence, it is very easy to imagine the following scenario (see Figure 1): two enterprises decide to become partners in a business process. One of the enterprises wants to buy something that the other enterprise offers, and to

achieve this, a sequence of messages has to be exchanged (e.g. purchase orders and purchase order acknowledgements). Each of these messages is represented in terms of the sender's ontology, and each of the business partners (e.g. enterprises) understands only messages expressed in terms of its own ontology. One of the roles of the execution environment (by mean of mediation), is to transform, if necessary, the received message from the terms of sender's ontology into the terms of the receiver's ontology, before sending it further. From the perspective of the ontologies, each message contains instances of the source ontology that have to be transformed into instances of the target ontology.

## 2.2. Addressed Ontologies

Web Service Modeling Ontology [WSMO, 2004] is a meta-ontology for describing several aspects related to Semantic Web Services. Its aim is to facilitate the usage of Web Services, providing the means for semi-automatic discovery, invocation, composition, execution, monitoring, mediation and compensation. For achieving this ambitious goal, WSMO defines four main modeling elements: Web Services, Goals, Ontologies and Mediators. The Web Services represent the functional part: each Web Service offers functionality which has to be semantically described, to enable its (semi-) automatic use. A Goal specifies the objective of a client: a client consults a Web Service only if this Web Service may satisfy his goal. The Ontologies have the role of providing semantics to the information used by all the other components, and the Mediators provide interoperability among the other components.

The purpose of this document is to present a possible approach for constructing a data mediation system able to mediate between data expressed in terms of WSMO ontologies. By working with WSMO compliant ontologies our data mediator can make full use of the fact that the input ontologies share the same meta-level: the WSMO conceptual model for ontologies.

## 2.3. Scope of Ontology Mediation in WSMX

The general scope of mediation in WSMX is similar to the scope of some other components part of WSMX architecture: to provide a default implementation of the desired functionality, in our case ontology-to-ontology mediation. The future intention is to provide support for integration of external components, providing the same functionality, but another (or better) realization of it.

As mentioned above, the WSMX data mediation addresses ontology-to-ontology mediation. The intention is to provide support for transforming instances referring to one ontology into instances specified in terms of the other ontology. The transformations applied to the source ontology affect both their structure and their values - the rules that drive these transformations may be seen as a mix of *integration* and *context* rules as described in [Wache, 1999]. The rules built based on the similar entities from the two ontologies (e.g. concepts and attributes) identified before runtime and saved in a storage internal to WSMX. The similar entities are defined and made known to WSMX by using a graphical user interface which allows the user to place his inputs and guides him/her through the entire process. Though similar entities are identified based on human inputs (choices, validations etc.), the rest of the process (rules generation and instances transformations) is carried out automatically, without human intervention.

## 3. Requirement Analysis

This section defines the main requirements for the mediation component in WSMX, and presents the main problems that need to be addressed during the ontology-to-ontology mediation process. In the first subsection, the functionality offered by the mediation component is analyzed in more detail, while in the second subsection a set of ontology mismatches is presented (as identified by [Visser et al., 1997]) relating them to our approach.

### 3.1. Ontology Mediation in WSMX

Ontology-to-ontology mediation in WSMX should provide three main functionalities: it should facilitate the identification of the potential similarities between the entities (e.g. concepts and attributes) of two given ontologies; it should provide the means of exploiting these similarities by creating appropriate rules to express the mappings between the two ontologies; and it should offer an environment in which the existing, mappings could drive the instance transformation process from the source to the target. In the following sections we will analyze the requirements for each of these functionalities.

#### Identifying similarities

This functionality can be seen as the first stage of the mediation process between two ontologies. Even if two ontologies are modeling the same aspects of the real world, they may have been developed by different parties, so different terminology might be used, or different modeling strategies might be chosen. As a consequence, we can infer the first requirement:

- The mediation process has to provide means to identify the similar entities from the two conceptualizations.

In fact, the process of finding similarities involves the identification of entities with the same semantic and unfortunately, even if different heuristics and techniques were developed for addressing this issue, the human user inputs are still necessary. Considering this:

- The mediation component has to offer an easy way to incorporate the domain expert inputs: a graphical interface that offers a visual representation of his or her actions and decisions.

Furthermore:

- This interface should include a set of the above-mentioned heuristics and techniques.

in order to reduce the human effort from a laborious and error-prone work to simple choices and validation decisions.

The graphical interface should provide facilities for:

- browsing the input ontologies;

- guiding the user through the whole process;
- hiding irrelevant information from the user;
- offering suggestions, based on the already-discovered similarities and other appropriate heuristics.

The user should always have access to the decisions they have already made for potential updates and refinements. After the process ends

- the discovered similarities have to be saved in a persistent storage for further usage

either for continuing later the similarity discovery process between the two ontologies or for applying them to the next stage of the mediation process. The way of storing these similarities should be independent of the language used for expressing the ontologies or for expressing the links in the next stage.

### Capturing the mappings

The next logical step that should be taken is to capture the already identified similarities into a formal description, to actually

- create the actual mappings between the two ontologies.

Each mapping should serve a well-defined purpose, which is to specify all the operations that have to be performed for transforming a set of instances from the terms of the first ontology into the terms of the second ontology. Such a mapping may contain specifications of the structural transformations of the source instance and/or of the values referred by that instance - transformations dictated by the structural differences between the ontologies and the differences between the contexts in which the instances are created [Wache, 1999].

These mappings may or may not be expressed in the same language as the ontologies, the decision of what language to use being determined by the complexity of the transformations required, and of the type of entities to be transformed. This stage of the mediation process has to be completely automatic:

- the links have to be created without any user or domain expert intervention.

This means that no interventions from the domain expert are required for this phase. Still, there could be the case that final adjustments are necessary, but in this phase, such adjustments have to be applied on the mapping rules directly, thus language specific knowledge are required from the human user side.

### Translating instances

The last step of the mediation process implies the actual transformations, and has to be performed based on the mappings created in the previous stage. That is, this stage (we may call it the execution engine) should provide

- means for executing the transformations specified in mappings

and for each given source instance a set of target instances has to be created. The execution engine is directly dependent on the language chosen in the previous stage for expressing the mappings and should be able to

- execute the transformations in a completely automatic way without any human intervention.

Based on the degree of involvement from the user side, we can say that the first functionality is to be implemented as part of the *design-time* phase, while the last two functionalities are to be made available in the *run-time* phase.

## 3.2. Ontology-to-Ontology Mediation Problems - Ontology Mismatches

In this section we present an exhaustive set of ontology mapping mismatches identified by [Visser et al., 1997] and analyze which of them should be addressed in our approach. The possible mismatches between ontologies are classified from two points of view: from the conceptualization point of view (*conceptualization mismatches*) and from the explications point of view (*explication mismatches*).

### Conceptualization mismatches

The conceptualization mismatches refer to differences between the ontological concepts and between the relations among these concepts. In the first case they consist of the *categorization mismatches* and the *aggregation level mismatches*, which are described in the following paragraphs.

*Categorization mismatch* occurs when the two conceptualizations distinguish the same concepts but organize them in different sub-concepts. A common example could be if one conceptualization is organized around the *mammals* and *birds* concepts and the other conceptualization around *carnivores* and *herbivores*. Our approach, considering the fact that our aim is to create for the given source instance a set of target instances, is not concerned with the mediation of sub-concept relationship between concepts, but only with the identification of the most similar concept from the target, for a given source concept.

The *aggregation-level mismatches* appear because of the usage of different levels of abstraction for defining similar concepts. As an example, one ontology can use the concept of *person* and another ontology can use the concepts of *male* and *female* without having a concept similar to person as super-concept. As in the previous case the scope of our mediation process seems to reduce the dimension of these types of mismatches.

Another conceptualization mismatch is the *relation mismatch* which is determined by the differences in the hierarchical relation between concepts and in the assignment of the attributes to the classes. There are three types of such mismatches identified: *structure mismatches*, *attribute-assignment mismatches* and *attribute-type mismatches*.

The *structure mismatches* are determined by the way the similar concepts are structured by means of relations. This version of mediation in WSMX doesn't address the mappings between the relations defined in the source and target ontology, even though the special case of binary relations is addressed in this

approach by considering their attributes. As a consequence, these mismatches, together with the *attribute-assignment mismatches* (generated by the different assignment of attributes to concepts) and *attribute-type mismatches* (occurring when two concepts have the same assigned attribute but differ by their assumed instantiation), are covered by the WSMX data mediation both at the stage of finding similarities and in the links creation process.

## Explication mismatches

[Visser et al., 1997] classify these mismatches based on the assumption that an ontology is formed by a set of definitions and each definition has the following form: Def=<T, D, C> where T stands for the term used, D for the definiens and C is the ontological concept that is explained. For two given definitions (one from the source ontology and one from the target ontology) there are eight ( $2^3$ ) possible combinations of mismatches, but there are two that are not relevant for this discussion: when all three terms are different (if there is no correspondence, we cannot talk about mismatch) or when all three terms are identical (there is no mismatch). All six remaining mismatches should be addressed by the WSMX data mediation at the similarity identifying stage, by being able to discriminate between cases where the two analyzed definitions refer to the same concept, or not. In fact, the task of identifying similarities involves the analysis of the terms (T), and of the definiens (D), and based on this analysis a conclusion has to be drawn: the explained concepts (C) are similar, or they are not similar. The six possible combinations of mismatches are discussed in more details below.

- **CT mismatch** - the modeled concepts and terms used are different but the definitions are the same. In terms of our approach, this means that we have the same definitions (the same attributes) for two concepts (with different names) that model different aspects of the domain. Even if the definitions are identical the concepts modeled are different - in this case the domain expert input should be considered for taking the correct decision.
- **CD mismatch** - the modeled concepts and the definitions are different but the terms used coincide; in our context, two different concepts are modeled using different attributes. Our assumption is that all concepts from the input ontologies are different until it is specified otherwise, so the system should be able to use the appropriate heuristics for inferring the correct suggestions (based on this suggestion the domain expert should be able to make the final decisions and validations). The terms used in this case are called homonyms.
- **C mismatch** - the modeled concepts are different but the definitions and terms used are identical. Here we have the same problem as in the CT mismatch. Domain expert intervention in this case is a must. Note that for domain-related ontologies this case should be very rare. As in the previous case the terms used are called homonyms.
- **TD mismatch** - the modeled concepts are similar but the terms used and the definitions are different. This means that two similar concepts are denoted by different terms and modeled using different attributes. At first sight, one may conclude that having a different definition for the two concepts the system will fail to make accurate suggestions. In reality, if the two input ontologies are modeling related domains, it is very likely that the definitions will converge on the same primitive elements (see Section 4.1 for more details related to the approach to this requirement). The terms used in this case are called synonyms.
- **T mismatch** - the modeled concepts and their definitions are similar but the

terms used are different. The offered heuristics should be able to suggest the two modeled concepts as similar (in this case the terms are also called synonyms). Based on this suggestions the domain expert should be able to make final decisions and validations.

- **D mismatch** - the modeled concepts and the terms used are the same but their definitions are different. The situation is very similar to the TD mismatch but the identical terms used should improve the quality of the returned suggestions.

The WSMX data mediation has to be able to respond to these ontology-to-ontology mediation problems. Some of these problems are eliminated by the nature of the problem we intend to solve, some of them should be addressed by means of the heuristics provided (heuristics results may be validated by the human expert) and others only by the explicit intervention of the human user in the similarity identification process.

## 4. WSMX Data Mediation Approach

[Wiederhold, 1994] identifies three main approaches that deal with ontology integration. The first of these approaches implies the use of domain experts for creating definitions for the terms in the used ontologies, definitions that must be accepted by all parties involved. When these definitions are completely documented and released, all the parties have to conform and adjust to these definitions. The second approach is to assume that the terms from the subject ontologies are matching and further discovered mismatches are resolved by annotating the corresponding terms with some source or domain identifiers. The last approach, in contrast, assumes that all the terms from the given ontologies are distinct and denote different concepts when not otherwise specified. This kind of information is explicitly stated using mapping rules.

The first approach is used in [Dou et al., 2003] in the design of the OntoMerge system used for ontology merging and automated reasoning. The system uses a merged ontology (containing symbols and facts from the source and target ontologies, together with a set of bridging axioms) for performing translations between two ontologies. But this approach is not suited to our needs, our aim being to create a tool that provides the mapping rules in a semi-automatic manner and then executes them.

The second approach could be useful for cases where the subject ontologies use very similar terminology for modeling the problem domain. Unfortunately, the ontologies are usually created independently, so there could be major differences even in representing the same domain. So, the best choice for our mediation component remains the third one: all the concepts from the source and the target ontology are different until mapping rules specify otherwise.

Following the requirements, our solution proposes three main steps for the WSMX data mediation: the *mapping creation* step for dealing with similarity identification, the *mapping rules generation* step for creating the links and the *execution* step for performing the instance translation. Each of these steps is presented in detail in the following sections.

## 5. Mappings

The mappings have the role of expressing the links between the source and the target ontology. Each of these links should express the similarity degree existing between the linked entities. In addition, the mappings represent the connection between the design-time and the run-time phases of the mediation process. As a consequence, we can state that the mappings represent the critical point of any mediator system, directly influencing the quality, effectiveness and efficiency of the mediation process.

In this section we discuss a way of representing this mappings as well as a methodology for their creation. We investigate then how the mappings act on the data to be transformed and provide a classification of the helper functions (known as conversion or built-in functions) together with strategies that can be used for their creation. This section ends with a description of the run-time phase of the mediation process, mainly of the rules generated from the mappings and their execution.

## 5.1. Abstract Mappings

As mentioned above, in our view, mappings represent the critical factor in any mediation process. Furthermore, the mappings could be the connecting point for different mediator systems that might share ontology mappings between them in order to cooperate in solving more difficult tasks. As a consequence, the mappings should be expressed in such a way that offer maximum reusability as well as great expressivity in order to accommodate a large number of mediation scenarios.

We chose for expressing our mappings to use a subset of the mapping language developed in SEKT project [de Bruijn et al., 2004]. Even if this language borrows some of the features of WSML-Flight [WSML, 2005] and OWL [Dean & Schreiber], it is not committed to any ontology representation formalism. Only an abstract syntax is provided and depending of its specific usage a formal semantics has to be associated to it. In our case, Section 9 provides a grounding of this abstract mapping language to Flora-2 [Flora-2]. Listing 1 presents the abstract syntax of the language subset used in our approach, written in EBNF. For more information about this mapping language please refer to [de Bruijn et al., 2004].

Listing 1. The abstract syntax of the mapping language subset (from [de Bruijn et al., 2004])

```

expression::='classMapping(['one-way'|'two-way']
                    classExprclassExpr
                    {attributeMapping}{classCondition}
                    [{'logicalExpression'}]')'

expression::='classAttributeMapping(['one-way'|'two-way']
                    (classExpr attributeExpr)|(attributeExpr classExpr)
                    {classAttributeMapping} {attributeMapping}
                    {classCondition} {attributeCondition}
                    [{'logicalExpression'}]')'

attributeMapping::='attributeMapping(['one-way'|'two-way']
                    attributeExpr attributeExpr
                    {attributeCondition}')'

classAttributeMapping::='classAttributeMapping(['one-way'|'two-way' ]
                    (classExprattributeExpr)|(attributeExprclassExpr)

```

```

        {classAttributeMapping} {attributeMapping}
        {attributeCondition} {classCondition}')'

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

attributeExpr ::= attributeID
    | 'and('attributeExpr attributeExpr {attributeExpr}')'
    | 'or('attributeExpr attributeExpr {attributeExpr}')'
    | 'not('attributeExpr')'
    | 'inverse('attributeExpr')'
    | 'symmetric('attributeExpr')'
    | 'reflexive('attributeExpr')'
    | 'trans('attributeExpr')'
    | 'join('attributeExpr attributeExpr {attributeExpr}
        [ {'logicalExpression'} ] )'

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

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

classID ::= URIReference
attributeID ::= URIReference
individualID ::= URIReference

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

```

Our approach adds another level of abstractions in order to guarantee the reusability and composition of the work done during the design-time phase. This additional abstraction level involves breaking the mapping rules in atomic entities that are to be discovered and stored as such in a persistent storage during design-time and retrieved and compose in rules during run-time. These atomic entities are classified in the following categories:

- **Concepts and attributes mappings.** They correspond to the simplest cases of *classMapping*, *attributeMapping* and *classAttributeMapping*:  
'classMapping(one-way' classExpr classExpr) ', 'attributeMapping(one-way' attributeExpr attributeExpr) ' and 'classAttributeMappings(one-way' classExpr attributeExpr) '. *[TO DO: discuss the possibility of defining two-way mappings]*. We consider in the next sections for the sake of simplicity the **classExpr** and **attributeExpr** as being simple *classIds* and *attributeIds* (in the future versions of this deliverable the possibility of having more complex expressions for classes and attributes will be considered). Section 5.2 describes how these mappings are derived during design-time.
- **Concept and attribute conditions.** They correspond to *classCondition* and *attributeCondition* respectively. They are derived together with concepts and attributes mapping during the design-time phase, as described

in [Section 5.2](#).

- **Conversion functions.** The conversion functions describe how the attributes value from the instances to be mediated are actually transformed. They correspond in the mapping language to logical expressions and in [Section 5.3](#) and [Section 5.4](#) you can find more details about their description and creation.

In the followings, we propose a way to create these atomic mapping entities, to store them in an efficient way and to compose them in complex mappings required by complex data transformations.

## 5.2. Mappings Creation

The mapping creation process represents one of the most important phases in a mediator system. It is a design time process and it is well known that in order to obtain high accuracy of the mappings the human user has to be present in this process. We believe that by offering a set of strategies and methodologies for creating this mappings we can reduce this error prone and laborious process from a manual task to truly semiautomatic one. The strategies and methodologies we proposed are not mapping patterns but they are applied on the source and target ontologies to identify the mapping language statements or existing mapping patterns.

In our view, the mapping process (i.e. the design time phase of the mediation) basically requires three types of actions from the domain expert:

- **Browse the ontologies:** The domain expert has to discover the ontology elements they are interested in. This step involves different views on the working ontologies, and strategies for reducing the amount of information to be processed by the human user (e.g. contexts based browsing).
- **Identify the similarities:** This step involves the identification of semantic relationships between the entities that are of interest in the two ontologies. For this the human user can make use of the suggestions offered by a set of lexical and structural algorithms for determining semantic relationships.
- **Create the mappings:** The last step involves the capturing of this semantic relationships by mappings. This means that the domain expert has to take the correct actions in order to properly catch the semantic relationships in the mapping language statements or maybe a predefined mapping patterns.

What we proposed in this section is a way of tackling the existing gap between the identified semantic relationships between the two ontologies and the mapping language (in our case a logical language) statements that captures this relationships. Mapping patterns can fill this gap only partially, the step from the semantic relationships to these patterns remaining uncovered.

This section describes how the input ontologies can be browsed by using different views, how the same ontological entities can play different roles in different views and how certain algorithms can be applied on these roles. We identified a set of views that can play an important role in the mapping creation process: *PartOf* view, *InstanceOf* view, *RelatedBy* view. Each of them will be described below.

### 5.2.1. General Strategies

We identify a set of general strategies that can be applied no matter what view is used for browsing the ontologies. Before describing these strategies we have to define several notions that will be used from now on:

- **View.** A *View* presents a subset of the ontology entities (e.g. concepts, attributes, relations, and instances) and the existing relationships between them. Usually the view used for browsing the source ontology (source view) and the view used for browsing the target ontology (target view) have the same type but there could be cases when different view types are used for source and target.
- **Roles.** In each of the views there is a predefined number of roles the ontology entities can have. In general, particular roles are fulfilled by different ontology entities in different views and in each of the strategies and algorithms described we refer to roles rather than ontology entities. The roles that can be identified in a view are: *Compound Item*, *Primitive Item*, *Description Item*.
- **Compound Item.** A *Compound Item* contains at least one description associated with it. For example in the *PartOf* view a compound item would be a concept that has at least one attribute (which plays the role of a Description Item).
- **Primitive Item.** A *Primitive Item* doesn't have any description associated. For example in the *PartOf* view this role is played by data types.
- **Description Item.** A *Description Item* offers more information about the Compound item it describes and usually links a Compound Item with other Compound or Primitive Items. By this we can define as *Successor* of a Description Item the Compound or Primitive Item it links to.

Figure 2 presents an abstract representation of a view and the main elements it consists of.

```

• primitive_item1
• compound_item1
  | hasDescription1 → compound_item2
  | hasDescription2 → primitive_item1
• primitive_item2
• primitive_item3
• compound_item2
  | hasDescription1 → primitive_item3
  | hasDescription2 → compound_item3
• compound_item3
  | hasDescription1 → primitive_item2
  | hasDescription2 → primitive_item1
  | hasDescription3 → primitive_item3

```

Figure 2. Abstract View

## Decomposition Algorithm

The decomposition algorithm is one of the most important algorithms in our approach and it is used for offering guidance to the domain expert in the mapping process. By decomposition we expose the descriptions of a compound item and make them available to the mapping process. That is, the decomposition

algorithm can be applied on description items and returns as result the description items (if any) for the successor of that particular description item. An overview of this algorithm is presented below:

```

decompose(Collection collectionOfItems) {
    Collection result;
    for each item in collectionOfItems do{
        if isCompound(item)
            Collection itemsDescriptions = getDescriptions(item);

            for each description in itemsDescriptions{
                Item successorItem = getSuccessor(description);

                if (not createsLoop(successorItem))
                    result = result + successorItem;
            }
    }
    return result;
}

```

The implementation of *isCompound*, *getDescriptions*, *getSuccessor*, and *createsLoop* differ from one view to another - for example the cases when loops are encountered (i.e. the algorithm will not terminate) have to be addressed for each view in particular.

## Contexts

During the mediation process not all the information available in the ontology is of interest for each particular phase of the mapping process. A *context* represents a subset of a view and presents only the relevant information for the current step of the mapping process. The notion of context goes hand in hand with the decomposition algorithm as a context is updated by applying this algorithm on a set of compound items. Thereby, by applying it recursively and updating the corresponding context, the domain expert is guided through the mapping process until all the items from the initial contexts were mapped.

Please note that when updating contexts the input of the human user has to be taken in consideration: the domain expert has to choose the source and the target items on which the decomposition process to be applied on. Of course, this choice can be done in a semi-automatic manner by suggesting the most probable source-target combinations to be further explored. Depending on the results returned by applying the decomposition algorithm on the source and on the target items respectively, four situations might be encountered:

- *Both sides decomposition*. For both the source and the target items the decomposition algorithm returned a non empty set of items. As a consequence both the source and the target contexts are updated.
- *One side decomposition - Source decomposition*. Only for the source items the decomposition returned a non empty set of items. This means that only the source context is updated while the target context remains unchanged.
- *One side decomposition - Target decomposition*. This is symmetric with the previous case. Only the target context is updated, the source context remaining unchanged.
- *No decomposition*. Successors were found neither for the source nor for the target, so no contexts can be updated. Usually this ends the decomposition and the mapping process for the current branches in the current source and target views.

## Suggestion Algorithms

The suggestion algorithms are used for helping the domain expert to take decision during the mapping process, regarding the possible semantic relationships between source and target items in the current context. Two types of such algorithms can be envisioned: the first one represented by the lexical based algorithms while the second type is represented by structural algorithms that consider the description item in their computations.

We propose a combination of lexical methods and strategies based on the description of the items to be mapped. As a result, for each pair of items we compute a so called *eligibilityfactor* (EF), which indicates the degree of similarity between the two items: the smallest value (0) means that the two items are completely different, while the greatest value (1) indicates that the two items are similar. For dealing with the values between 0 and 1 a threshold value is used - the values lower than this value indicate different items and values greater than this value indicate similar elements. Setting a lower threshold assures a greater number of suggestions, while a higher value for the threshold restricts the number of suggestion to a smaller subset. The EF is computed as an weighted average between a *structural factor* (SF), referring to the structural properties and a *lexical factor* (LF), referring to the lexical relationships determined for a given pair of items.

Even if the structural factor is computed using the decomposition algorithm, the actual heuristics used are dependent on the specific views where it is applied. In a similar manner the current views determine the weight for the structural and lexical factors as well as the exact features of the items to be used in computations.

## Bottom-Up vs Top-Down Approach

Considering the algorithms and methods described above two possible approaches regarding ontology mapping can be differentiated: a bottom-up and a top-down approach.

The bottom-up approach means that the mapping process starts with the mappings of the primitive items (if possible) and then continues with items having more and more complex descriptions. By this, the primitive items act like a minimal, agreed upon set between the two ontologies, and starting from this minimal set more complex relationships can be gradually discovered.

The top-down approach implies that the mapping process starts directly with mappings of compound items and it is usually adopted when a concrete heterogeneity problems has to be resolved. That means the domain expert is interested only in resolving particular items mismatches and not in fully align the input ontologies. The decomposition algorithms and the contexts it updates will help them to identify all the relationships that can be captured by using that type of view and are relevant to the problems to be solved.

In the same manner as for the other algorithms, the applicability and advantages/disadvantages of each of these approaches is dependent on the type of view used.

### 5.2.2. PartOf View

The *PartOf* is probably the most popular view on the ontologies to be align. The roles of *Primitive items* is taken by the *primitive concepts* (i.e. data types) while the role of *Compound items* is taken by *concepts* described by at least one attribute. The *descriptions* are represented by *attributes* and naturally, the *successor* of a description is the *range* of that attribute. As shown in Figure 3 the successor of a description in this view (i.e. the range of an attribute) can be either a primitive concept or a compound concept. Using this view we can create the following set of mappings:

```

• primitive_concept (data type)
• compound_concept
  | attribute1 → primitive_concept (range)
  | attribute2 → compound_concept (range)

```

Figure 3. Elements of *PartOf* View

- **Primitive Concept to Primitive Concept mappings.** This mapping generates a *classMapping* statement in the abstract mapping language.
- **Primitive Concept to Compound Concept mappings.** The *PartOf* view does not allow these type of mappings. Such a situation is covered by mapping a compound concept from the source that has an attribute with the primitive concept as range with a compound concept from the target.
- **Compound Concept to Primitive Concept mappings.** These is a situation identical with the above one and the *PartOf* view does not allow these type of mappings as well. The rational behind this restriction comes from the fact that such combinations would generate artificial mappings (with no semantics) between primitive concepts and all the compound concepts that refer by means of their attributes to these primitive concepts. For example, any compound concept that has an attribute with the range *String* could be mapped with *String*.
- **Compound Concept to Compound Concept mappings.** This mapping generates a *classMapping* statement in the abstract mapping language corresponding to the two compound concepts and triggers the decomposition mechanism, followed by a set of mappings between the attributes of these compound concepts, respectively. Such mappings between attributes are described below.
- **Attribute to Attribute mappings.** There are four cases that can be encountered in this situations, generated by the two types of concepts an attribute can have as range: primitive concept or compound concept (i.e. primitive range or compound range).
  - *Primitive range* on the source and *primitive range* on the target.  
An *attributeMapping* is generated in the abstract mapping language followed by a mapping between two primitive concepts.
  - *Primitive range* on the source and *compound range* on the target.  
This case generates in the abstract mapping language a *classAttributeMapping* between the owner of the source attribute and target attribute followed by a mapping between two compound concepts: the owner of the source attribute and the range of the target attribute.
  - *Compound range* on the source and *primitive range* on the target.  
This case is symmetric with the one presented above and it generates a *classAttributeMapping* in the abstract mapping

- language (actually this is an *attributeClassMapping* but there is only one statement in the language for both situations) and leads to a compound concept to compound concept mapping as well.
- *Compound range* on the source and *compound range* on the target. An *attributeMapping* is generated in the abstract mapping language followed by a mapping between two compound concepts.

### 5.2.3. *InstanceOf* View

During the modelling process a set of instances can be used to properly capture some of the features of the domain. This is the case for enumeration sets containing for example geographical locations, categories or even the allowed values for certain data-types (e.g. true and false for boolean). In the *InstanceOf* view the primitive items' role is taken by such instances (we call them *primitive instances*). By using these primitive instances more complex instances (*compound instances*) could be created, that is, instances of compound concepts whose attributes have ranges for which primitive or compound instances already exist or can be created. The compound instances play the role of compound items in this view (see Figure 4). The descriptions for the compound instances are represented by the attributes and attribute values corresponding to the compound instances. The attribute values are in fact the successors of compound items descriptions.

```

• primitive_instance
• compound_instance
  | attribute1 → primitive_instance
  | attribute2 → compound_instance

```

Figure 4. Elements of *InstanceOf* View

*InstanceOf* view is used for creating conditional mappings and almost all the cases presented in the *PartOf* view occur in this view as well but with the difference that conditions are associated to mappings:

- **Primitive Instance to Primitive Instance mapping.** This mapping generates a *instanceMapping* statement in the abstract mapping language.
- **Primitive Instance to Compound Instance mapping.** Mappings between a primitive and a compound instance are not allowed in the *InstanceOf* view from similar reasons as in the *PartOf* view.
- **Compound Instance to Primitive Instance mapping.** The same restriction applies as above.
- **Compound Instance to Compound Instance mapping.** This mapping generates a *classMapping* statement in the abstract mapping language corresponding to the two compound concepts that own the two compound instances and triggers the decomposition mechanism, followed by a set of mappings between the attribute values of these compound instances, respectively. Such mappings between attributes' values are described below.
- **Attribute value to Attribute value mapping.** There are four cases that can be encountered in this situation, generated by the two types of instances an attribute can have as value: primitive instance or compound instance (i.e. primitive instance range or compound instance range).
  - *Primitive instance range* on source and *primitive instance range* on target.

An *attributeMapping* is generated in the abstract mapping language conditioned by two *attributeValueConditions* - one for the source and the other one for the target attribute.

- *Primitive instance range* on source and *compound instance range* on target.

This case generates in the abstract mapping language a *classAttributeMapping* between the owners of the source attribute and target attribute followed by a mapping between two compound instances: the owner of the source attribute and the range of the target attribute. In addition a *typeCondition* on the target attribute is applied.

- *Compound instance range* on source and *primitive instance range* on target.

This case is symmetric with the one presented above and it generates a *classAttributeMapping* in the abstract mapping language and leads to a compound instance to compound instance mapping as well. In addition a *typeCondition* on the source attribute is applied.

- *Compound instance range* on source and *compound instance range* on target.

An *attributeMapping* and two *typeConditions* one for the source and the other one for the target attribute, are generated in the abstract mapping language followed by a mapping between two compound instances.

#### 5.2.4. *RelatedBy* View

[TO BE WRITTEN]

### 5.3. Conversion Functions

[TO BE WRITTEN]

### 5.4. Conversion Functions Creation

[TO BE WRITTEN]

### 5.5. Mapping Rules Generation

After the schema level mappings between the two ontologies are done, these mappings are used to create the *mapping rules*. A mapping rule specifies the way in which the values in the first instance are extracted and used for the creation of the target instance(s). This stage in the mapping process is necessary in order to provide strong decoupling between the form of representing the mappings and a particular ontology representation language. This approach offers a set of advantages as: it simplifies the mappings maintenance and management, it offers flexibility in choosing the appropriate grounding language in conformance with each specific application requirements and it makes from these abstract mappings an important point of alignment between different mediation systems.

As a consequence, in this step the grounding mechanism has to be applied on the abstract mappings and to retrieve the mapping rules expressed in a language

dependent format. The language the mapping rules are generated in, is usually determined by the language in which the input ontologies and source instance are expressed; also the new target instance will be expressed in the same language. Some translators may be used to allow the usage of different languages (e.g. for expressing the target instance in another language).

Once a mapping rule is created, it can be saved for other potential usages or it can be redone each time this is necessary. If the used ontologies are unlikely to change, then it may be more efficient to store the mapping rules in a persistent storage and just reuse them when they are needed. If the ontologies are evolving, and they are frequently changing, the second approach would be more appropriate: consistency must be maintained only between the mappings in order to have consistent mapping rules.

## 5.6. Execution

This step deals with the execution of the existing mapping rules using an environment that understands the language in which the source instance and the rules are represented. The result of the execution could be one or more instances of concept(s) from the target ontology. This step may also include the checking of the potential constraints defined in the source or in the target ontology and which affect the mapped elements. That is, using the execution environment one can check the correctness and the consistency of the mapping rules.

## 6. Addressed Ontology Mismatches

*[THIS SECTION IS SLIGHTLY OUTDATED - TO BE EXTENDED]*

In this section, the set of problems that are addressed in our approach is presented. In the future, this set of problems will be extended and the future versions of this deliverable and of the provided prototype will offer solutions for a greater number of possible mismatches that can appear during the mapping process.

### 6.1. Different Composition of Concepts

This is the most common problem that can appear during the mapping process and it is caused by the fact that concepts that share the same semantic meaning are modeled in different ways, e.g. using different numbers and types of attributes and relations. That is, we can have different levels of conceptualizations used for modeling the same aspect of the domain, even if in the end, at the lowest level, we have the same primitive concepts. This problem is also described as granularity of match or simply as granularity in [Chalupsky, 2000].

This problem is addressed by applying the decomposition process, revealing the internal structure of the concepts, and recursively moving from one level to another until the primitive elements are encountered. What remains to be done is to create the direct mappings between these primitive elements; we will see later that this could be in some case a non-trivial task at all.

### 6.2. Incomplete/Over-complete Concept Representation

Another problem that may appear during the mapping process is that a required concept (either from the source or from the target ontology), does not have a similar concept in the other ontology. In this case a decision has to be made regarding the missing concept and this decision mostly depends on whether the missing concept appears in the source or in the target ontology. In the first case, the data will be mediated without any content loss (even if from the target point of view the data will be incomplete), but in the second case some of the content will be lost in the mediation process (but from the target point of view the data will be complete).

In general, for this kind of problem several solutions could be adopted, depending on the requirements of the two business partners. When the missing concept is on the source side the mapping rule must assure that in the created instance (target instance) a default value is filled in. Also there are some cases when simply ignoring the missing concept could be enough, the responsibility for dealing with the missing value being delegated to the application that is using the new created instances. When the missing concept is on the target side, the problem is more complicated. Normally, the target doesn't know what to do with an instance of a concept that it cannot understand, but there may be some situations in which the target will have to return back that instance, for example, as a participation proof in the previous communication. So, a solution could be to simply discard, during the mapping rule execution, all the instances of an unmapped concept (assuming that there will be no need for these instances in the future) or mapping them as instances of a bag concept whose only role is to gather all unmapped instances.

The simplest solution is adopted for the first version of this deliverable: when the missing concept is in the source ontology no instance of the target concept is created; when the missing concept is in the target the corresponding source instance (or value) is discarded.

### 6.3. Data-Type Mismatch

Another common problem for the mediation process appears when concepts denoting the same aspects are defined using different data types. The most common example is when the data-type for an attribute is *string* in one of the ontologies, and for the similar one in the other ontology a more specific data type is used, like *integer*.

We could have different dimensions of this problem, depending of the casting compatibility of the two data types involved. In the above example the solution is very simple when the attribute having as data type an *integer* is part of the source ontology (*integer* can always be converted to a *string*) but if we consider it the other way around, the solution is not trivial at all, since not any *string* can be converted to an *integer*.

Considering our approach, this problem can appear only in the final phase of the mediation process, when mapping between primitive concepts, after all the required decompositions have already been done. Between the attributes, the mappings can be simple assignments from the source attributes to the target attributes, but also more complicated formulas that make the explicit conversion from one data format to the other. The checking of whether the mapping rule has specified the correct operation for converting from one data format to the other is actually done only when the rule is executed. The operations performing the

required transformations can be seen as the context rules, described in [[Wache, 1999](#)].

## 6.4. Cardinality

When creating the mappings between ontologies, the following situation may also occur: some concepts from one of the ontologies are subsumed by one concept from the other ontology. In [[Omelayenko & Fensel, 2001](#)] this class of problems is referred as being 1:n, n:1 and n:m mapping cases and in [[Rahm & Bernstein, 2001](#)] it is considered to be part of the match cardinality process as set-oriented mapping case.

It is important to mention here that this problem appears only when mapping primitive elements, so no other concept (and implicitly instance) details are available. The cardinality problem complexity is dependent on where the subsumed concepts are placed in the source ontology or in the target ontology. In the first case, one can get from the subsumed concepts to the target concept by applying a certain operation (e.g. concatenation). In the second case it is not so trivial to create a mapping rule that splits a given instance (which does not have an explicitly defined structure) in a set of subsumed concepts instances. The only solution for this situation is to find out details about the source instance's internal structure from other information sources (for example the instance provider). The most general case of subsumption is the n:m mapping, but it can be seen as m cases of 1:n mappings.

The problem presented in the above section is a particular case of the cardinality problem, from the point of view of the adopted solution. That is, for this case, we have to provide operations that are able to accept as input one or more source instances and to return one or more target instances. But one has to consider the fact that each of the owners might create their instances in their own way and it is the role of the mediation process to make the required transformations.

## 7. Flora-2 - A reasoner for WSMX Data Mediation

As described in Section 5.5 mapping rules are created based on the mappings created during the design-time phase and stored in given repository. These mapping rules are language dependent and their task is to enable the appropriate reasoner to execute the actual instance transformations. The language to express the mapping rules is usually chosen as being the language in which the input ontologies are expressed. As our scope is WSMO, the ontology language we assume our ontology (in most of the cases) would be expressed in WSML [[WSML, 2005](#)]. Unfortunately at the time this document is written no WSML reasoner is available. As a consequence, a reasoner compatible with our requirement had to be found and used as a replacement until a WSML reasoner will be available. As discussed in [[WSML Reasoner, 2004](#)] Flora-2 [[Flora-2](#)] is a good candidate for reasoning with different WSML variants and also fulfills the needs of the data mediator. [*TO DO: Would be nice to elaborate this a little bit*].

Having Flora-2 as a reasoner for WSMX Data Mediator has the followings consequences: transformations between Flora-2 and WSML (and back) and grounding from the abstract mapping language to Flora-2 are required. The next two subsections, 7.1 and 7.2, describe in more details these aspects, respectively.

## 7.1. From WSML to Flora-2

The WSMX Data Mediation component requires transformation between WSML and Flora-2 only for the runtime phase. That is, during design-time WSML ontologies can be loaded directly and the mappings are expressed in an abstract way. But during run-time the incoming instances (data to be mediated) is expressed in WSML and they have to be processed by the Flora-2 reasoner. For making this possible these instances have to be transformed first in Flora-2 instances and only then loaded in the Flora-2 environment. After the mediation process ends the mediated data (target ontology instances) come out from the reasoner as Flora-2 instances. Before being delivered further for other processing to WSMX they have to be transformed in WSML instances.

As a consequence we can say that our main point of interest in WSML to Flora-2 transformation are the instance transformations. We envision that in the future version of this data mediator the reasoner might be used for other tasks as well (e.g. mapping validations against the source and target ontologies), so transformations between WSML and Flora-2 syntax would be required for the other ontology elements too. [Appendix A2](#) describes how WSML to Flora-2 transformations looks like when recursive transformation functions are used.

## 7.2. Expressing Mapping Rules in Flora-2

The mapping language we use is an abstract one and it doesn't have any formal semantics. By creating a grounding to a concrete language, formal semantics is provided and in addition we can benefit from the reasoning support available for the language we ground to.

In this section we propose a grounding to Flora-2 for the abstract mapping language described in [de Bruijn et al., 2004], having as model the grounding to WSML-Flight as presented in [Predoiu et al., 2004]. We construct transformation functions for the mapping language statements to Flora-2, having the following form:

$$T(\text{mapping\_language\_element}) \rightarrow \text{Flora-2\_statement}$$

where *mapping\_language\_elements* represents syntactical elements of the mapping language as described in the EBNF grammar in Listing 1 and *Flora-2\_statement* represents a Flora-2 syntactical construction. Recursive calls are allowed and by applying these transformation functions on the mapping language statements Flora-2 rules are obtained. The strings between single quotations signs appears as such in the mapping language or in Flora-2 (depending on which side of the  $\rightarrow$  signs are placed) and the strings without quotations play the role of variables or elements that can be decomposed according with the EBNF grammar in Listing 1. Please note that in the following listings the constructs as

$$T(\text{argument}) \rightarrow \text{result}$$

don't have anything to do with any kind of rules from particular logical formalism; the semantics of these construct is that by applying the function *T* on the given *argument* a particular *result* is obtained.

Due to the relation of the abstract mapping language with WSML-Flight [[WSML](#),

2005], when defining some of the grounding functions to Flora-2 we make use of some transformations functions defined in Appendix A2 (i.e. transformations function between WSMML and Flora-2). In order to avoid confusions the grounding function is called  $T$ , while for the transformation functions between WSMML and Flora-2 the  $t$  symbol is used.

Listing 3. Grounding general elements of the language

```

T(URIReference) → t('<"URIReference">')

T(typedLiteral) → t(typedLiteral)

T(plainLiteral) → t(plainLiteral)

T(logicalExpression) → t(logicalExpression)
T(logicalExpression, Y) → t(logicalExpression[X:=Y])

```

We have two different transformations for the logical expression because it could be the case that variables used inside the logical expression to be syntactically substituted during the translation process (e.g. the variable  $X$  inside of the logical expression to be substituted by  $Y$ ).

Listing 4. Grounding the class mappings

```

T('classMapping(two-way' classExpr1 classExpr2
  attributeMapping1...attributeMappingn
  classCondition1...classConditionm
  logicalExpression1...logicalExpressionq')) →
  T('classMapping(one-way' classExpr1 classExpr2
    attributeMapping1...attributeMappingn
    classCondition1...classConditionm
    logicalExpression1...logicalExpressionq'))
  T('classMapping(one-way' classExpr2 classExpr1
    attributeMapping1...attributeMappingn
    classCondition1...classConditionm
    logicalExpression1...logicalExpressionq'))

T('classMapping(one-way' classExpr1 classExpr2
  attributeMapping1...attributeMappingn
  classCondition1...classConditionm
  logicalExpression1...logicalExpressionq')) →
  T(classExpr2, mediated(X)) ':-' T(classExpr1, X)', '
  T(classCondition1, X)', '
  ...
  T(classConditionp, X)', '
  T(classConditionp+1, mediated(X))', '
  ...
  T(classConditionm, mediated(X))', '
  T(logicalExpression1, X, mediated(X))', '

```

```

...
T(logicalExpressionq1, X, mediated(X))'.'
T(attributeMapping1)'.'
...
T(attributeMappingn1)'.'
T(attributeMappingn1+1, logicalExpressionq1+1)'.'
...
T(attributeMappingn, logicalExpressionq)'.'

```

$1 \leq p \leq m$   
 $1 \leq q_1 \leq q$   
 $1 \leq n_1 \leq n$

The two-way class mappings can be expressed as two one-way class mappings. The extra parameter used in the one-way transformation function for class expression is the variable which will be used in the Flora-2 rule (i.e. Flora-2 rules are obtained by applying the transformation functions on mapping language statements) to represent an instance of the given class expression. This variable is also used for linking the class mapping itself with the class conditions, attribute mappings and logical expression. Please note that two sets of class conditions are identified in this transformation, the first one applied on the source and the others on the target. We also have two sets of logical expressions, classified after their usage: they are used to introduce axioms that refer in the first set to the classes that are mapped (by the mean of source and target instance) and in the second set to the attributes that are mapped. For the second set an example of such axioms are the conversion functions, used for transforming a source attribute value according to the specified function.

Another important point in the class mapping transformations is that the target instance (the mediated instance) name is built using a function symbol (i.e. *mediated()*) and its properties (e.g. the concept it belongs to, the attributes and their values) are imposed by the rules generated by the transformation functions. A class mapping statement transformation usually generates more than one Flora-2 rule: at least one for the class mapping itself, class conditions and logical expressions, and at least one for attribute mappings and the corresponding logical expressions.

Listing 5. Grounding the class expressions

```

T('and('classExpr1 ... classExprn')', X) →
    T(classExpr1, X) ',' ... ',' T(classExprn, X)

T('or('classExpr1 ... classExprn')', X) →
    '(' T(classExpr1, X) ';' ... ';' T(classExprn, X) ')'

T('not('classExpr')', X) →
    'not(' T(classExpr, X) ')'

T('join('classExpr1 ... classExprn
    logicalExpression1...logicalExpressionq')', f(X2, ... ,Xn)) →
    T(classExpr1, f(X2, ... ,Xn))

T(classID, X) →

```

```
X ':' T(classID)
```

Please note that class expressions containing disjunction (*or*) and negation (*not*, which stands for negation based on well-founded semantics - for more details see [Flora-2]) may be used only as a source in class mappings because disjunction and negation is not allowed in the rules head (i.e. the transformations generate Flora-2 rules that have to conform to this restriction). If a class expression containing *join* is encountered, the transformations in Listing 6 have to be considered in addition to the rules generated by the transformations in Listing 4.

Listing 6. Auxiliary transformations for class expressions containing *join*

```
T(classExpr1, f(X2, ... , Xn) ':' -'
  T(classExpr2, X2) ',' ... ',' T(classExprn, Xn) ',' ,
  T(logicalExpression1, {f(X2, ... , Xn), X2, ... , Xn}) ',' ... ',' ,
  T(logicalExpressionq, {f(X2, ... , Xn), X2, ... , Xn}) ':' .'
```

Listing 7 presents the transformation functions for grounding the attribute mappings. As in the case of the class mappings the two-way attribute mappings are decomposed in two one-way attribute mappings. For the one-way mappings two transformation functions are defined, depending on the way the attributes values are passed from the source to the target instance as follows:

- the first function generates two Flora-2 rules:
  - the first rule triggers when the attribute value is an instance and creates for the target attribute value a new instance using the function symbol *mediate()* (this instance will be recursively described by other rules generated by the transformation functions).
  - the second rule triggers when the attribute value is a literal and passes the value from the source to the target unchanged.
- the second function takes in account a logical expression that may specify how the value from the source is transformed before being passed to the target. Although the logical expression may look like the first rule generated by the first function, we recommend to use this function when the source attribute value is a literal.

Listing 7. Grounding the attribute mappings

```
T('attributeMapping(two-way' attributeExpr1 attributeExpr2
  attributeCondition1...attributeConditionn')') →
  T('attributeMapping(one-way' attributeExpr1 attributeExpr2
    attributeCondition1...attributeConditionn')')
  T('attributeMapping(one-way' attributeExpr2 attributeExpr1
    attributeCondition1...attributeConditionn')')
T('attributeMapping(one-way' attributeExpr1 attributeExpr2
  attributeCondition1...attributeConditionn')') →
  T(attributeExpr2, mediated(X), mediated(Y)) ':' -'
    mediated(Y) ':' -' ',' T(attributeExpr1, X, Y) ',' ,
  T(attributeCondition1, mediated(X), attributeExpr2)
```

```

', ' ...
T(attributeConditionp, mediated(X), attributeExpr2), '
T(attributeConditionp+1, X, attributeExpr1)
', ' ...
T(attributeConditionn, X, attributeExpr1).'
T(attributeExpr2, mediated(X), Y) :-
\+'mediated(Y)':_ ' ', ' T(attributeExpr1, X, Y)', '
T(attributeCondition1, mediated(X), attributeExpr2)
', ' ...
T(attributeConditionp, mediated(X), attributeExpr2), '
T(attributeConditionp+1, X, attributeExpr1)
', ' ...
T(attributeConditionn, X, attributeExpr1).'

T('attributeMapping(one-way'attributeExpr1attributeExpr2
attributeCondition1...attributeConditionn)', logicalExpression) →
T(attributeExpr2, mediated(X), Y1) :- 'T(attributeExpr1, X, Y2)', '
T(attributeCondition1, mediated(X), attributeExpr2)
', ' ...
T(attributeConditionp, mediated(X), attributeExpr2), '
T(attributeConditionp+1, X, attributeExpr1)
', ' ...
T(attributeConditionn, X, attributeExpr1), '
T(logicalExpression, Y1, Y2).'

```

1<=p<=n

In the same way as for the class conditions, the attribute condition may be applied on the source attributes or on the target attributes.

The transformations for the attribute mappings (Listing 8) contain two more additional parameters. The first parameter is a variable representing the instance the attribute is part of and the second parameter represents the value of the attribute for that particular instance.

Listing 8. Grounding the attribute expressions

```

T(attributeID, X, Y) → X['T(attributeID) '->' Y]'

T('and('attributeExpr1...attributeExprn)', X, Y) →
T(attributeExpr1, X, Y)', '...', 'T(attributeExprn, X, Y)

T('or('attributeExpr1...attributeExprn)', X, Y) →
('T(attributeExpr1, X, Y)'; '...' ; 'T(attributeExprn, X, Y)')

T('not('attributeExpr')', X, Y) →
'not'T(attributeExpr, X, Y)

T('inverse('attributeExpr')', X, Y) →
T(attributeExpr, Y, X)

T('symetric('attributeExpr')', X, Y) →

```

```

T(attributeExpr, X, Y)

T('reflexive('attributeExpr')', X, Y) →
  T(attributeExpr, X, Y)

T('trans('attributeExpr')', X, Y) →
  T(attributeExpr, X, Y)

```

[TO DO: To associate the specific semantics to *inverse*, *symmetric*, *reflexive* and *transitive* attributes (for now they are just ignored. ]

The class conditions transformation function in Listing 9 has an additional parameter representing the instance that owns the attribute which the condition is applied on.

Listing 9. Transformations for class conditions

```

T('attributeValueCondition('attributeID individualID')', X) →
  X['T(attributeID) '->' T(individualID)']

T('attributeValueCondition('attributeID dataLiteral')', X) →
  X['T(attributeID) '->' T(dataLiteral)']

T('attributeTypeCondition('attributeID classExpr')', X) →
  X['T(attributeID) '->' Y'] ' ', ' T(classExpr, Y)

T('attributeOccurrenceCondition('attributeID')', X) →
  X['T(attributeID) '=>' _']

```

In Listing 10 the attribute conditions are transformed by using a transformation function with two additional parameters: the first one represents the owner of the attribute on which value the conditioned is applied and the second one the attribute itself. Please note that the attribute we mentioned above might be an attribute expression as well.

Listing 10. Transformations for attribute conditions

```

T('valueCondition('individualID')', X, Y) →
  T(Y, X, T(individualID))

T('valueCondition('dataLiteral')', X, Y) →
  T(Y, X, T(dataLiteral))

T('typeCondition('classExpr')', X, Y) →
  T(Y, X, Z) ' ', ' T(classExpr, Z)

```

## 8. Prototype Implementation

[THIS SECTION IS SLIGHTLY OUTDATED - TO BE UPDATED]

As described above, the mediation process we propose consists of three main steps: *mapping generation*, *mapping rules creation* and *execution*. Accordingly, the mediation module will consist of three main sub-modules or components: a graphical user interface for defining the mappings, a component that reads the mappings from the storage and generates the appropriate mapping rules and an

environment that provides the means for executing the mapping rules. Figure 5 shows how these components interact.

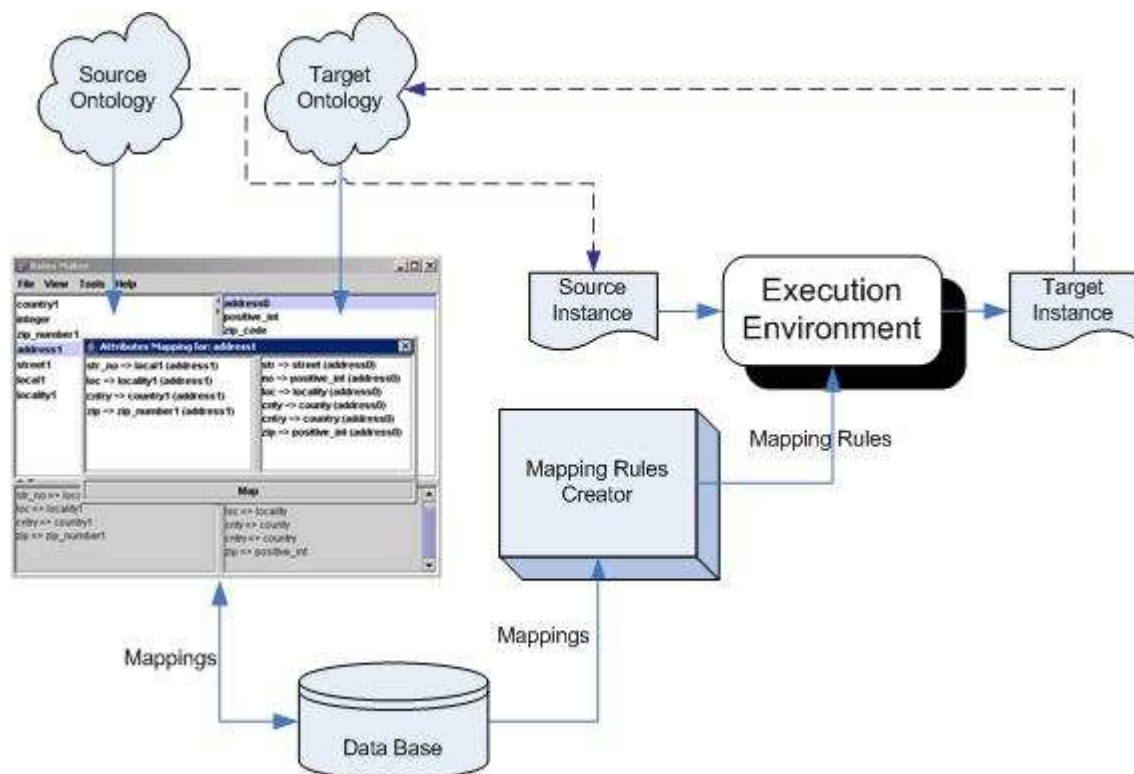


Figure 5. Overview of the Data Mediation Module

## 8.1. Graphical User Interface

The mapping creation process cannot be performed 100% automatically and as a consequence, the domain expert has to provide inputs at various stages of this process. Of course, the mappings could be done entirely manually, but this has proven to be a very laborious, error-prone and time-consuming process. Considering this, the mapping tools offer graphical interfaces in order to assist the user in the mapping creation and to reduce his or her effort to simple choices and validations. These kinds of graphical mapping tools are also described in [Maedche et al., 2002] and [Noy & Musen, 2000].

The graphical component described here has two main roles: firstly to guide the user to the mapping creation process, and secondly, to compute the suggestions based on already done direct and/or indirect mappings. The first task is accomplished by presenting the user the current source and target contexts containing the eligible elements for mapping. In fact, in this top-down approach, the user knows what elements s/he wants to map and through the decomposition process the contexts are updated in each step guiding the user to the primitive elements that will be mapped in the last step. This top-down approach may be very easily combined with a bottom-up approach (this being the recommended solution), in order to enable the suggestion mechanism to return better results. That is, the user can address in the beginning the mapping of some of the basic elements (including primitive elements), in this way constructing a basic vocabulary which will be referred in the next top-down mappings.

The second task is achieved by computing a similarity factor between the

elements to be mapped. This similarity measure is based on two similarity factors: the first one is a heuristic function that computes an eligibility factor based on the already done mappings (direct or indirect mappings), and the second one is a lexical similarity factor based on the syntactical similitude between the name of the elements and their nonfunctional properties. These two factors are used together with the decomposition process in order to compute the similarities between compound elements as well as between primitive elements. It is important to notice that without establishing a basic set of mappings between primitive and basic elements of the two ontologies, the results returned by the first factor, the eligibility measure, cannot be accurate. For this, the mappings between primitive concepts could be determined by using the lexical factor, together with a thesaurus (e.g. WordNet).

The graphical interface is built in Java and is able to load Flora-2 [Flora-2] ontologies. The created mappings are stored in an external storage, in this case a relational database, from where they can be loaded by the mapping rules generator module. Also, by means of this graphical interface, the user can load already existing mappings from the external storage for further refinements or as support in computing the suggestions.

## 8.2. Mapping Rules Generator Module

This module deals with the creation of the mapping rules and expresses them in a format that can be later executed. It takes as input a concept from the source ontology and an identifier of the target ontology and searches the storage to find mappings between the given concept and concepts from the target ontology. There are three situations that can occur: no mappings to target concepts are found, exactly one mapping to a concept from the target is found, or several mappings to different concepts from the target ontology are identified.

In the first case the creation of rule fails and an error message is returned. This means that a return to the previous step is required in order to provide suitable mappings. The second case, when only one mapping exists, is the easiest one to solve: the decomposition process is applied in order to consider all involved elements (attributes and concepts). But the third case is a little bit more complicated. Having more than one mapping could mean that there are several concepts in the target that are similar to the concept in the source ontology. Going further, and assuming that the mappings are correct, this means that in the target, some equivalent concepts are defined which usually is not a good modeling choice, unless this is explicitly specified (e.g. by inheritance relationships, or by subsumptions and aggregations). We will consider that the input ontologies are correctly modeled, so the source concept can be mapped to more equivalent concepts with a different degree of generality. As a consequence, a decomposition process is applied and a similarity factor is computed based on the indirect and direct mappings implied by the initial ones. Note that even if the target ontology is badly modeled and there are equivalent concepts without an explicit relation between them, the above described process will pick one, and due to these equivalences the choice will be as good as any other.

The generated mapping rules will be taken by the execution environment and optionally they could be stored for performance improvements or caching facilities. In our case the mapping rules are represented in Flora-2 and no storing actions are performed.

## 8.3. Execution Environment

As mentioned above, the mapping rules are expressed in Flora so the execution environment is a Flora-2 environment for XSB [Flora-2]. This environment is wrapped by a Java interface and it is ready to take the source instance and the mapping rule and to return the target instance(s) by executing the rule. The target instance is also expressed in Flora but an XML description for it may be provided. For this version no other computations are done in this environment, but it could be used also for testing the validity of the mappings and to perform constraints checking.

The last two sub-modules are used by WSMX at runtime to provide and to execute the mapping rules. For each source instance provided by WSMX, the available mappings are retrieved from the storage and a mapping rule is created. The newly created mapping rule is forwarded to the execution module, executed, and the result (i.e. the new target instance) is returned to WSMX. The graphical user interface was used earlier on, before runtime, for defining and storing the mappings between similar entities.

The first version of this prototype allows the user (more or less a domain expert) to create a set of mappings between concepts and attributes of two ontologies. The types of mismatches that can be addressed and solved using this prototype were described in sections 4.2.1, 4.2.2, 4.2.3, and 4.2.4. The process of creating mappings could be seen as an iterative one: the user should analyze the suggestion offered, implement some of them and then re-explore the new set of suggestions offered (the suggestions are strongly affected by the created mappings: better suggestions are obtained based on a higher number of mappings). Once the mapping process has ended, the mappings can be dropped in the external storage to be picked by the run-time mediation component, transformed in mapping rules and executed in order to obtain the target instances.

## 9. Related Work

In this section we present two related approaches to ontology to ontology mediation. We chose to present PROMPT [Noy & Musen, 2000] and MAFRA [Maedche et al., 2002], because of their similarities to our approach both in the conceptual framework proposed and in the software implementation they offer.

### 9.1. PROMPT

PROMPT is an algorithm and a tool proposed by Noy and Musen [Noy & Musen, 2000] which allows semi-automated ontology merging and alignment. It takes as inputs two ontologies and guides the user through an iterative process for obtaining a merged ontology as an output. This process starts with the identification of the classes with similar names and provides a list with initial matches. Then the following steps are repeated several times: the user selects an action (by choosing a suggestion or by editing the merged ontology directly) and the tool computes new suggestions and determines the eventual conflicts. This iterative process is presented in Figure 6.

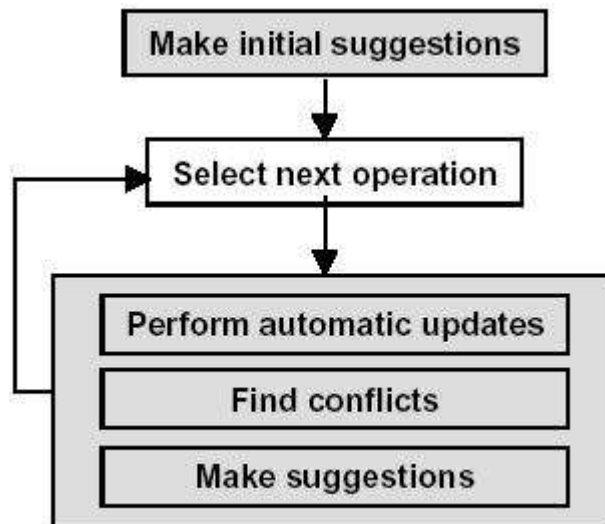


Figure 6. The PROMPT Algorithm

PROMPT is able to handle ontologies conforming to OKBC [Chaudri et al., 1998] format having at the top level classes, slots, facets, and instances. The operations allowed include merge classes, merge slots, merge binding between a slot and a class, perform deep/shallow copy of a class from one ontology to another (including/not including all the parents of the class up to the root of hierarchy and all the classes and slots it refers to). Some of the conflicts identified by PROMPT are name conflicts, dangling references (a frame refers to another frame that doesn't exist), redundancy in the class hierarchy (more than one path from a class to a parent other than root) or slot-value restriction that violates class inheritance.

### Implementation

PROMPT was implemented as a plugin in Protege-2000 - a knowledge base acquisition tool, and by this PROMPT can take advantage of all the ontology engineering capabilities of this tool. Some of the features offered by this implementation are: setting the preferred ontology (for automatic conflict-solving in favor of one ontology), maintaining the user's focus (suggestions the user sees first should relate with the frame in the same area), providing explanations for the suggestions made, logging and reapplying operations.

## 9.2. MAFRA

MAFRA [Maedche et al., 2002] is a Mapping Framework for Distributed Ontologies, designed to offer support at all stages of the ontology mapping life-cycle. Also, a partial implementation is offered for this framework as part of the Kaon Ontology and Semantic Web Framework.

The architecture of the MAFRA conceptual framework is presented in Figure 7.

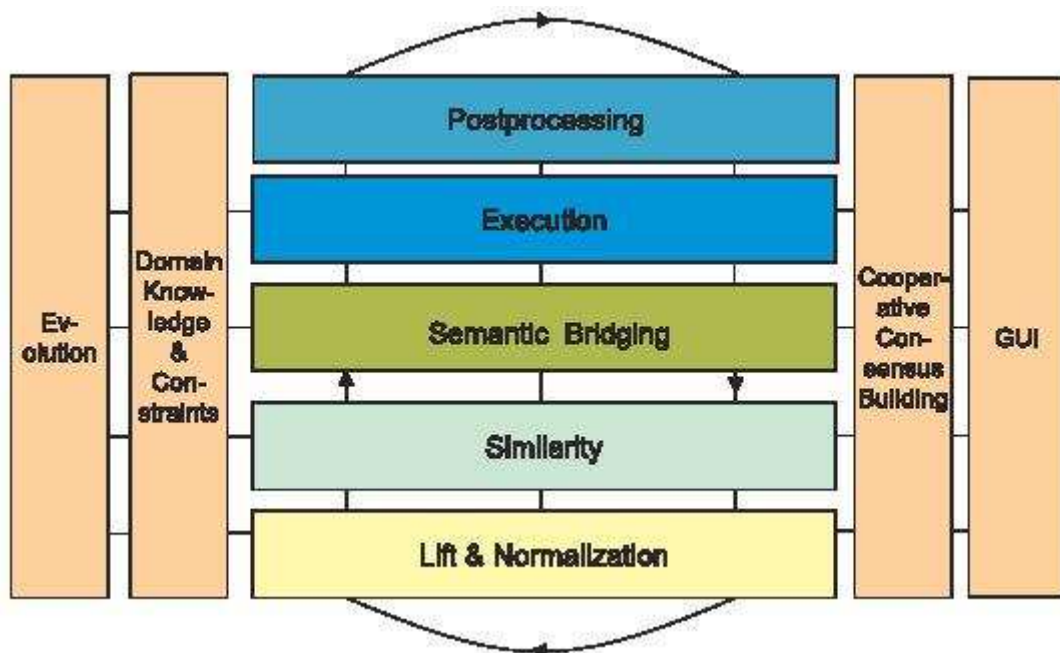


Figure 7. Conceptual Architecture

The framework is organized in two dimensions: it contains horizontal and vertical modules. The horizontal modules (*Lift & Normalization*, *Similarity*, *Semantic Bridging*, *Execution*, *Post-processing*) describe fundamental and distinct phases in the mapping process, while the vertical modules (*Evolution*, *Domain Constraints & Background Knowledge*, *Cooperative Consensus Building*, *Graphical User Interface*) run along the entire mapping process interacting with the horizontal modules.

### Horizontal Dimension

The *Lift & Normalization* module has the role of coping with syntactical, structural and language heterogeneity, raising all the data to be mapped to the same representation language. Both ontologies are normalized to a uniform representation, in this case RDF(S). The *Similarity* module implements strategies for discovering the similarities between entities. It determines lexical similarity, as well as so called property similarities - similarities between concepts based on their properties, either attributes or relations. Also it defines bottom-up/top-down similarities for propagating the similarities or dissimilarities from lower/upper parts of the taxonomy to the upper/lower concepts. *Semantic bridging*, based on the similarities previously determined, establishes correspondences between entities from the source and target ontologies. It specifies bridges between entities in a way that each instance represented according to the source ontology is translated into the most similar instance described according to the target ontology. The *Execution* module actually transforms instances from the source ontology to the target ontology, by evaluating the semantic bridges defined in the previous step, and the *Post-processing* module takes the results to check and improve the quality of the transformations.

### Vertical Dimension

*Evolution* focuses on keeping the bridges obtained by the Semantic Bridging module up to date, consistent with the transformation and the evolutions that may take place in the source and target ontologies. *Cooperative Consensus Building* is

responsible for establishing a consensus between two communities relative to the semantic bridges used. The need for this module comes from the multiple bridges available for performing transformations and from the aim of reducing the human contribution in the mapping process. *Domain Constraints & Background Knowledge* may improve substantially the quality of bridges by using background knowledge and domain constraints, as glossaries, lexical ontologies or thesauri. Finally, the *Graphical User Interface* is required due to the fact that the mapping creation is a difficult and a time consuming process, thus extensive graphical support must be provided.

## 10. Conclusions and Further Directions

This deliverable provides solutions for ontology-to-ontology mediation problems, as a starting-point for the more complex problem of mediation in the context of Semantic Web Services. Our intentions are to provide conceptual foundations for a semi-automatic approach to ontology-to-ontology mediation, together with a software implementation as a testbed and a proof of their validity. Also, the software implementation is integrated in Web Service Execution Environment (WSMX) as a default implementation of the mediation component.

In our future work, the focus will be on improving the solutions proposed for ontology mediation, including the extension of the problem set we address, together with the creation of better heuristics for computing suggestions. Another aim is to further explore the mediation problems in the context of WSMO, offering solutions for the other types of mediators which were not addressed in this document and integrating their implementation into the WSMX mediation component.

## References

**[Chalupsky, 2000]** H. Chalupsky: OntoMorph: A Translation System for Symbolic Knowledge. In *Proceedings of 7th International Conference on Knowledge Representation and Reasoning (KR), Breckenridge, (CO US), pages 471-482, 2000.*

**[Chaudhri et al., 1998]** V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, J. P. Rice: OKBC: A Programmatic Foundation for Knowledge Base Interoperability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)* (pp. 600-607). Madison, Wisconsin, USA: MIT Press, 1998.

**[de Bruijn et al., 2004]** J. de Bruijn, Douglas Foxvog, Kerstin Zimmerman: Ontology mediation patterns. SEKT Project Deliverable D4.3.1, Digital Enterprise Research Institute, University of Innsbruck, 2004.

**[Dean & Schreiber]** M. Dean and G. Schreiber (eds.): OWL Web Ontology Language Reference. 2004. W3C Recommendation 10 February 2004.

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

**[Dou et al., 2003]** D. Dou, D. McDermott, P. Qi: Ontology Translation on the Semantic Web. In *ODBASE'03*, 2003.

**[Fensel & Bussler, 2002]** D. Fensel, C. Bussler: The Web Service Modeling Framework (WSMF). In *White Paper and Internal Report Vrije Universiteit Amsterdam*, 2002.

**[Flora-2]** Available at <http://flora.sourceforge.net/>

**[Lin et al., 2001]** H.Lin, T. Risch T. Katchaounov: Adaptive Data Mediation Over XML Data. In *Journal of Applied System Studies (JASS)*, Cambridge International Science Publishing, 2001.

**[Madhavan et al., 2002]** J.Madhavan , P. A. Bernstein , P. Domingos , A. Y. Halevy: Representing and Reasoning About Mappings Between Domain Models. *Eighteenth National Conference on Artificial intelligence*, p.80-86, Edmonton, Alberta, Canada, July 28-August 01, 2002.

**[Maedche et al., 2002]** A. Maedche, B. Motik, N. Silva, R. Volz: MAFRA - A Mapping Framework for Distributed Ontologies. In *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW*, Madrid, Spain, September 2002.

**[Mitra et al., 2000]** P. Mitra, G. Wiederhold, M. Kersten: A Graph-Oriented Model for Articulation of Ontology Interdependencies. In *Proceeding Extending DataBase Technologies, Lecture Notes in Computer Science*, vol. 1777, pp. 86-100, Spreinger, Berlin Heidelberg New York, 2000.

**[Noy & Musen, 2000]** N. Noy, M. Musen. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2000.

**[Omelayenko & Fensel, 2001]** B. Omelayenko, D. Fensel: An Analysis of Integration Problems of XML-Based Catalogues for B2B E-commerce. In *Proceedings of the 9th IFIP 2.6 Working Conference on Database (DS-9)*, Semantic Issues in e-commerce Systems, Hong Kong, April 2001.

**[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/>.

**[Rahm & Bernstein, 2001]** E. Rahm, P. Bernstein: A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal* 10: 334-350, 2001.

**[Visser et al., 1997]** P. R. S. Visser, D. M. Jones, T. J. M. Bench-Capon, M. J. R. Shave: An Analysis of Ontological Mismatches: Heterogeneity Versus Interoperability. In *AAAI 1997 Spring Symposium on Ontological Engineering*, Stanford, USA, 1997.

**[Wache, 1999]** H. Wache: Towards Rule-based Context Transformation in Mediators. In *S. Conrad, W. Hasselbring, and G. Saake, editors, International Workshop on Engineering Federated Information Systems (EFIS 99)*, Kuhlungsborn, Germany, 1999.

**[Wache et al., 2001]** H. Wache, T. Vogele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, S. Hubner: *Ontology-Based Integration of Information - A Survey of Existing Approaches*. In *Proceedings of IJCAI-01 Workshop: Ontologies and Information Sharing*, Vol. pp. 108-117, Seattle, WA, 2001.

**[Wiederhold, 1994]** G. Wiederhold: *Interoperation, Mediation, and Ontologies*, In *Proceedings International Symposium on Fifth Generation Computer Systems (FGCS94), Workshop on heterogeneous Cooperative Knowledge-Bases*, Vol.W3, pages 33-48, Tokyo, Japan, Dec. 1994.

**[WSML, 2005]** J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, D. Fensel: *The WSML Family of Representation Languages*, WSML Working Draft v0.2, 2005, available at <http://www.wsmo.org/TR/d16/d16.1/v0.2/>

**[WSML Reasoner, 2004]** J. de Bruijn, C. Feier, Uwe Keller, R. Lara, A. Polleres, L. Predoiu: *WSML Reasoner Implementation*, WSML Working Draft v0.2, 2004, available at <http://www.wsmo.org/2004/d16/d16.2/v0.2/>

**[WSMO, 2004]** D. Roman, U. Keller, H. Lausen (eds.): *Web Service Modeling Ontology*, version 1.0, 2004, available at <http://www.wsmo.org/2004/d2/v1.0/>

**[WSMX, 2005]** E. Cimpian, M. Moran, E. Oren, T. Vitvar, Michal Zaremba: *Overview and Scope of WSMX*, WSMX Working Draft v0.2, 2005, available at <http://www.wsmo.org/TR/d13/d13.0/v0.2/>

**[WSMX Execution Semantics, 2005]** Maciej Zaremba, E. Oren: *WSMX Execution Semantics*, WSMO Working Draft v0.1, 2004, available at <http://www.wsmo.org/2004/d13/d13.2/v0.1/>

**[WSMX Architecture, 2005]** Michal Zaremba, M. Moran: *WSMX Architecture*, WSMO Working Draft v0.2, 2005, available at <http://www.wsmo.org/2005/d13/d13.4/v0.2/>

## Acknowledgments

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

The editors would like to thank all the members of the WSMO working group for their advice and inputs to this document.

---

## Appendix A. WSML-Core to Flora-2 Translation

### A1. The corresponding mappings between WSML-Core and Flora-2

|  |  |  |
|--|--|--|
|  |  |  |
|--|--|--|

| WSML-Core conceptual syntax                                                                                                                      | Flora-2 syntax                                                                   | Remarks |
|--------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|---------|
| <i>Logical Declarations</i>                                                                                                                      |                                                                                  |         |
| <b>instance</b> <i>o</i> <b>memberOf</b> <i>C1, ..., Cn</i><br><i>A1</i> <b>hasValue</b> <i>V1</i><br>...<br><i>Am</i> <b>hasValue</b> <i>Vm</i> | <i>o:C1. ... o:Cn</i><br><br><i>o[A1 -&gt; V1,</i><br>...<br><i>An -&gt; Vn]</i> |         |

Table 2: Mapping between WSML-Core and Flora-2 syntax

## A2. Translating WSML-Core abstract syntax into Flora-2 syntax by using a recursive translation function

### General

```

//id
//id - iri
t('<' iri_reference '>') →
  "' iri_reference '"

t(ncname1 ':' ncname2) →
  t(ncname2)

t('_' ncnamechar1 ... ncnamecharn) →
  '_' ncnamechar1 ... ncnamecharn

t([0x0041 ... 0x005A] ncnamechar1 ... ncnamecharn) →
  "[0x0041 ... 0x005A] ncnamechar1 ... ncnamecharn"

t([0x0061 ... 0x007A] ncnamechar1 ... ncnamecharn) →
  [0x0061 ... 0x007A] ncnamechar1 ... ncnamecharn

//id - anonymous
t(anonymous) →
  anonymous

//id - literal
t(plainliteral '^' iri) →
  t(plainliteral)

t('' literal_content '') →
  "'literal_content'"

t(number) →
  number

```

$t(\text{string}) \rightarrow$   
string

$t(\text{'true'}) \rightarrow$   
'true'

$t(\text{'false'}) \rightarrow$   
'false'

## NonFunctionalProperties

//nfp  
 $t(\text{'nfp' attributevalue}_1 \dots \text{attributevalue}_n \text{'endnfp'}) \rightarrow$   
 'nonFunctionalProperties' '->' '\_#' '['  $t(\text{attributevalue}_1)$  ',' ... ','  $t(\text{attributevalue}_n)$   
 ']'

$t(\text{'nonFunctionalProperties' attributevalue}_1 \dots \text{attributevalue}_n$   
 $\text{'endNonFunctionalProperties'}) \rightarrow$   
 'nonFunctionalProperties' '->' '\_#' '['  $t(\text{attributevalue}_1)$  ',' ... ','  $t(\text{attributevalue}_n)$   
 ']'

## Concepts

//concepts  
 $t(\text{'concept' id superconcept nfp attribute}_1 \dots \text{attribute}_n) \rightarrow$   
 $t(\text{'concept' id superconcept})$  ','  $t(\text{'concept' id nfp attribute}_1 \dots \text{attribute}_n)$

$t(\text{'concept' id superconcept nfp}) \rightarrow$   
 $t(\text{'concept' id superconcept})$  ','  $t(\text{'concept' id nfp})$

$t(\text{'concept' id superconcept attribute}_1 \dots \text{attribute}_n) \rightarrow$   
 $t(\text{'concept' id superconcept})$  ','  $t(\text{'concept' id attribute}_1 \dots \text{attribute}_n)$

$t(\text{'concept' id superconcept}) \rightarrow$   
 $t(\text{superconcept, id})$

$t(\text{'concept' id nfp attribute}_1 \dots \text{attribute}_n) \rightarrow$   
 $t(\text{'concept' id nfp})$  ','  $t(\text{'concept' id attribute}_1 \dots \text{attribute}_n)$

$t(\text{'concept' id nfp}) \rightarrow$   
 $t(\text{id})$  '[' 'nonFunctionalProperties' '->'  $t(\text{nfp})$  ']'

$t(\text{'concept' id attribute}_1 \dots \text{attribute}_n) \rightarrow$   
 $t(\text{id})$  '['  $t(\text{attribute}_1)$  ',' ... ','  
 $t(\text{attribute}_n)$  ']'

$t(\text{'concept' id}) \rightarrow$   
 $t(\text{id})$

```
//superconcept
t('subConceptOf' '{ id ',' id1 ',' ... ',' idn }', X) →
    t('subConceptOf id, X) ',' t('subConceptOf id1, X) ',' ... ',' t('subConceptOf idn,
X)

t('subConceptOf id, X) →
    X '::' t(id)
```

## Attributes

```
//attributes
t(id1 'ofType' id2) →
    t(id1) '=>' t(id2)

t(id1 'ofType' id2 nfp) →
    t(id1) '[' 'nonFunctionalProperties' '->' t(nfp) ']' '=>' t(id2)

t(id1 'impliesType' id2) →
    t(id1) '=>' t(id2)

t(id1 'impliesType' id2 nfp) →
    t(id1) '[' 'nonFunctionalProperties' '->' t(nfp) ']' '=>' t(id2)

//the following rule should be added when 'impliesType' is used: 'X:Y :- _[A => Y],
_[A -> X].'
```

```
//attribute values
t(id1 'hasValue' id2) →
    t(id1) '->' t(id2)

t(id 'hasValue' '{ id0 ',' id1 ',' ... ',' idn }') →
    t(id) '->>' '{ t(id0) ',' t(id1) ',' ... ',' t(idn) }'
```

## Instances

```
//instances
t('instance' id memberof nfp attributevalue1 ... attributvaluen) →
    t(memberof, id) '.' t('instance' id nfp attributevalue1 ... attributvaluen)

t('instance' id memberof nfp) →
    t(memberof, id) '.' t('instance' id nfp)

t('instance' id memberof attributevalue1 ... attributvaluen) →
    t(memberof, id) '.' t('instance' id attributevalue1 ... attributvaluen)

t('instance' id memberof) →
    t(memberof, id)
```

$t(\text{'instance' id nfp attributevalue}_1 \dots \text{attributvalue}_n) \rightarrow$   
 $t(\text{'instance' id nfp}) \text{' , ' } t(\text{'instance' id attributevalue}_1 \dots \text{attributvalue}_n)$

$t(\text{'instance' id nfp}) \rightarrow$   
 $t(\text{id}) \text{' [ 'nonFunctionalProperties' '->' } t(\text{nfp}) \text{' ]'}$

$t(\text{'instance' id attributevalue}_1 \dots \text{attributvalue}_n) \rightarrow$   
 $t(\text{id}) \text{' [ ' } t(\text{attributevalue}_1) \text{' ; ' ... ' ; '}$   
 $t(\text{attributevalue}_n) \text{' ]'}$

$t(\text{'instance' id}) \rightarrow$   
 $t(\text{id})$

//memberof  
 $t(\text{memberof, X}) \rightarrow$   
 $t(\text{'memberOf' idlist, X})$

$t(\text{'memberOf' '{' id ' ; ' id}_1 \text{' ; ' ... ' ; ' id}_n \text{' } , X) \rightarrow$   
 $t(\text{'memberOf' id, X}) \text{' ; ' } t(\text{'memberOf' id}_1, X) \text{' ; ' ... ' ; ' } t(\text{'memberOf' id}_n, X)$

$t(\text{'memeberOf' id, X}) \rightarrow$   
 $X \text{' ; ' } t(\text{id})$

## Axioms

$t(\text{'axiom' id}) \rightarrow$   
 $t(\text{id}) \text{' ; '}$

$t(\text{'axiom' id nfp 'definedBy' log\_expr}) \rightarrow$   
 $t(\text{id}) \text{' [ 'nonFunctionalProperties' '->' } t(\text{nfp}) \text{' ; '}$   
 $\text{' definedBy -> } \{ \{ t(\text{log\_expr}) \} \text{' ]'}$

$t(\text{'axiom' id 'definedBy' log\_expr}) \rightarrow$   
 $t(\text{id}) \text{' [ 'definedBy -> } \{ \{ t(\text{log\_expr}) \} \text{' ]'}$

$t(\text{'axiom' nfp 'definedBy' log\_expr}) \rightarrow$   
 $\_ \# [ \text{'nonFunctionalProperties' '->' } t(\text{nfp}) \text{' ; '}$   
 $\text{' definedBy -> } \{ \{ t(\text{log\_expr}) \} \text{' ]'}$

$t(\text{'axiom' 'definedBy' log\_expr}) \rightarrow$   
 $\_ \# [ \text{'definedBy -> } \{ \{ t(\text{log\_expr}) \} \text{' ]'}$

## Logical expressions

$t(\text{expr ' . '}) \rightarrow$   
 $t(\text{expr})$

$t(\text{expr '<- ' disjunction}) \rightarrow$   
 $t(\text{expr}) \text{' :- ' } t(\text{disjunction}) \text{' ; '}$

$t(\text{expr } \rightarrow \text{ disjunction}) \rightarrow$   
 $t(\text{disjunction}) \text{ ':' } t(\text{expr}) \text{ '}'$

$t(\text{expr } \leftarrow \text{ disjunction}) \rightarrow$   
 $t(\text{expr}) \text{ ':' } t(\text{disjunction}) \text{ '}'$   
 $t(\text{disjunction}) \text{ ':' } t(\text{expr}) \text{ '}'$

$t(\text{expr } @ \text{ iri}) \rightarrow$   
 $t(\text{expr}) @ t(\text{iri})$

$t(\text{disjunction } \text{or} \text{ conjunction}) \rightarrow$   
 $t(\text{disjunction}) \text{ ';' } t(\text{conjunction}) \text{ '}'$

$t(\text{conjunction } \text{and} \text{ subexpr}) \rightarrow$   
 $t(\text{conjunction}) \text{ ';' } t(\text{subexpr}) \text{ '}'$

$t(\text{'neg' subexpr}) \rightarrow$   
 $\text{'not' } t(\text{subexpr}) \text{ '}'$

$t(\text{'not' subexpr}) \rightarrow$   
 $\text{'not' } t(\text{subexpr}) \text{ '}'$

$t(\text{'(' expr ')'}) \rightarrow$   
 $\text{'(' } t(\text{expr}) \text{ ')}'$

## Appendix B. Changelog

2005.05.16

- Minor updates through the whole document (e.g. fixing typos etc)
- Small updates on *Section 3*
- Updates of *Subsection 5.2*

2005.03.21

- Content was added in *Section 5.2.1*

2005.03.07

- *Section 7.2*: First version of the grounding of the abstract mapping language to Flora-2
- Updates in *Appendix A*

2005.02.18

- Changes in the *Introduction* section: the WSMO-Lite references were removed and the document overview rewritten, and the out of date references updated
- *Section 2.2* was shortened: the listings containing the old WSMO ontological elements were removed
- Listing 1 in *Section 5.1* was updated: conceptAttributeMapping statement was added

- Content was added in *Section 7*
- *Refereces* section was adjusted: the unused references were removed and couple of new ones were added.
- *Appendix A2* updated: transformation functions for concepts and attributes were added

2005.01.28

- This changelog was added
- The Appendix A, containing an initial version of WSML-Core to Flora-2 translation was added. This first version covers only the instance transformations from WSML to Flora-2.
- The introduction in section 5 as well as subsection 5.1



webmaster