D13.3v0.2 WSMX Data Mediation

WSMX Working Draft 11 October 2005

Final version:
http://www.wsmo.org/TR/d13/d13.3/v0.2/20051011
Latest version:
http://www.wsmo.org/TR/d13/d13.3/v0.2/
Previous version:
http://www.wsmo.org/TR/d13/d13.3/v0.2/20050708
Editor:
Adrian Mocan
Authors:
Adrian Mocan
Emilia Cimpian
Reviewer:
Jos de Bruijn

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

Abstract

Data heterogeneity problems have been investigated intensely in the last decade, in an attempt to find solutions to data base integration, query rewriting, exchanged business documents transformations, product catalogs management etc. Unfortunately, no complete or general solutions were found for these problems, so data mediation still remains a trade-off between the desired degree of accuracy and the amount of human involvement needed in the process. In this document we investigate data mediation as part of the emerging semantic technologies for Semantic Web and Semantic Web Services. Particularly we try to see how ontologies can be used in this process, and what tool support is required in order to increase the degree of automation provided and to reduce the human user effort. For this we describe the concrete mechanisms both for the design-time phase of the mediation process that lead to a semi-automatic creation of alignments between ontologies and for the run-time phase which applies in an automatic way the alignments derived during design time.

Glossary

Alignment: Set of mappings between ontologies having the role of reconciling the overlapping subsets of the two ontologies.

Compound Item: A compound item is an item having at least one description.

Context: A context represents a subset of a view and presents only the relevant information for the current step of the mapping process. The contexts can be seen as horizontal subsets of a view - a given set of ontological entities and relationships from a view are presented.

Description Item: An item describing a compound item (sometimes called just description).

Execution Semantics: Execution semantics is the formal definition of the operational behaviour of a system and describes in a formal, unambiguous language how the system behaves. The formalism used by [WSMX Execution Semantics, 2005] to model the execution semantics is Petri-nets.

Item: Graphical representation of a role in a view tree.

Mappings: These express semantic relationships between two or more elements of ontologies. A mapping can refer to conditional statements that have to hold for the semantic relationship to hold. Such mappings are represented in abstract form and they don't have any formal semantics associated with them. In this document the mappings are represented in the abstract mapping language proposed in [de Bruijn et al., 2004].

Mapping Rules: The mapping rules are obtained by grounding the abstract mappings to a concrete ontology representation language. In this document, we refer to Flora-2 mapping rules obtained by applying the Flora-2 grounding on the abstract mappings.

Ontology Mediation: The process of creating an alignment between different ontologies.

Primitive Item: An item having no description associated.

Role: A placeholder for ontology entities that is used in general algorithms and strategies.

Successor: The primitive or compound item which the description points to.

View: A subset of the ontology entities and the relationships existing between them. The views can be seen as vertical subsets of the ontology – all the entities and relationships of a specific type (dictated by each particular view) from the ontology are considered.

Table of Contents

---

1. Introduction

This document 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, 2005], 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.

1.1. General Considerations

A range of different models or ontologies describing the same or related problem domains could be created by different entities throughout the world. This implies that 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, such as *Information Integration and Semantic Web*, *Data Migration* or *Ontology Merging* [Madhavan et al., 2002].

Unfortunately, there is always a trade-off between how accurate these mappings are and the degree of automation that can be offered. There are approaches able to provide these kinds of mappings (also known as alignments) between different schemas or ontologies using machine learning techniques ([Doan et al., 2002], [Euzenat et al., 2004]) in an automatic manner but only with limited accuracy. In order to rule out the false results, the domain expert has to validate and check the mappings or the alignment at the end of the process. Another type of approach considers the human intervention from the beginning, proposing an interactive mapping process where the tool suggestions and the human user validations alternate in the process until the final result is achieved ([Noy & Musen, 2000], [Maedche et al., 2002]).

The data mediation solution presented in this document follows the second approach described above: we propose well-defined strategies and methodologies for the mapping process in order to guarantee - the most correct and complete mappings possible, together with a set of algorithms and strategies meant to make the mapping task much easier (reducing it to simple validations and choices). We adopted this approach because we believe that in the context of Web Services and business transactions the transformations on data must be 100% accurate. In addition, we consider that an interactive approach towards mapping creation is much more appropriate in the case of medium/large ontologies and also when the intention is to abstract the domain expert (using a graphical interface) from the underlying logical formalism used to represent the mappings.

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.

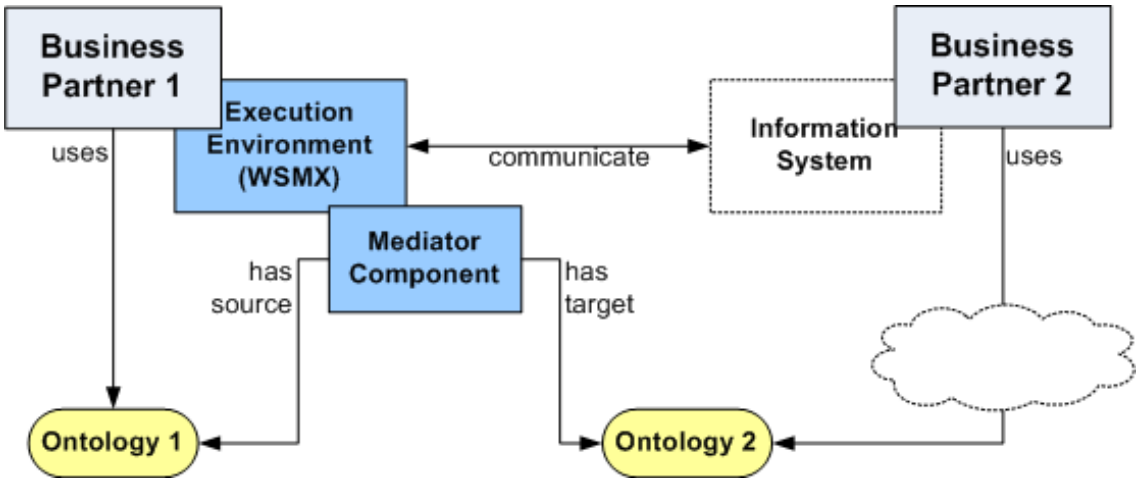


Figure 1. WSMX Data Mediation Scenario

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. The default ontology language we consider is Web Service Modeling Language (WSML) [WSML, 2005], but we not exclude the possibility of creating transformations from/to another ontology languages as OWL [Dean & Schreiber] for example (such transformations to and from OWL DL are already defined in [WSML, 2005]).

2.3. Scope of Ontology Mediation in WSMX

The general scope of data mediation in WSMX identifies with the scope of some other components part of WSMX architecture: to provide a default implementation for a specific piece of functionality, in our case ontology-to-ontology mediation. The intention is to provide support for integration of external components, providing the same functionality, but another (or better) realization of it. For this, the first step is to provide well defined interfaces that have to implement by any external component that intent to add its functionality to the WSMX architecture.

As mentioned above, 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. Other use cases of ontology mediation, like ontology merging for example, are part of the scope of this work. The execution semantics identified by [WSMX Execution Semantics, 2005] for data mediation is quite simple (see Figure 2):

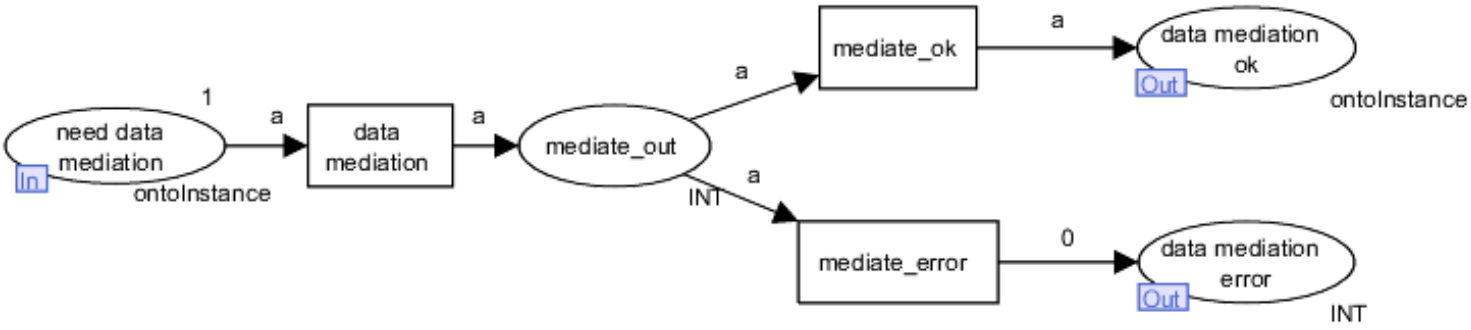
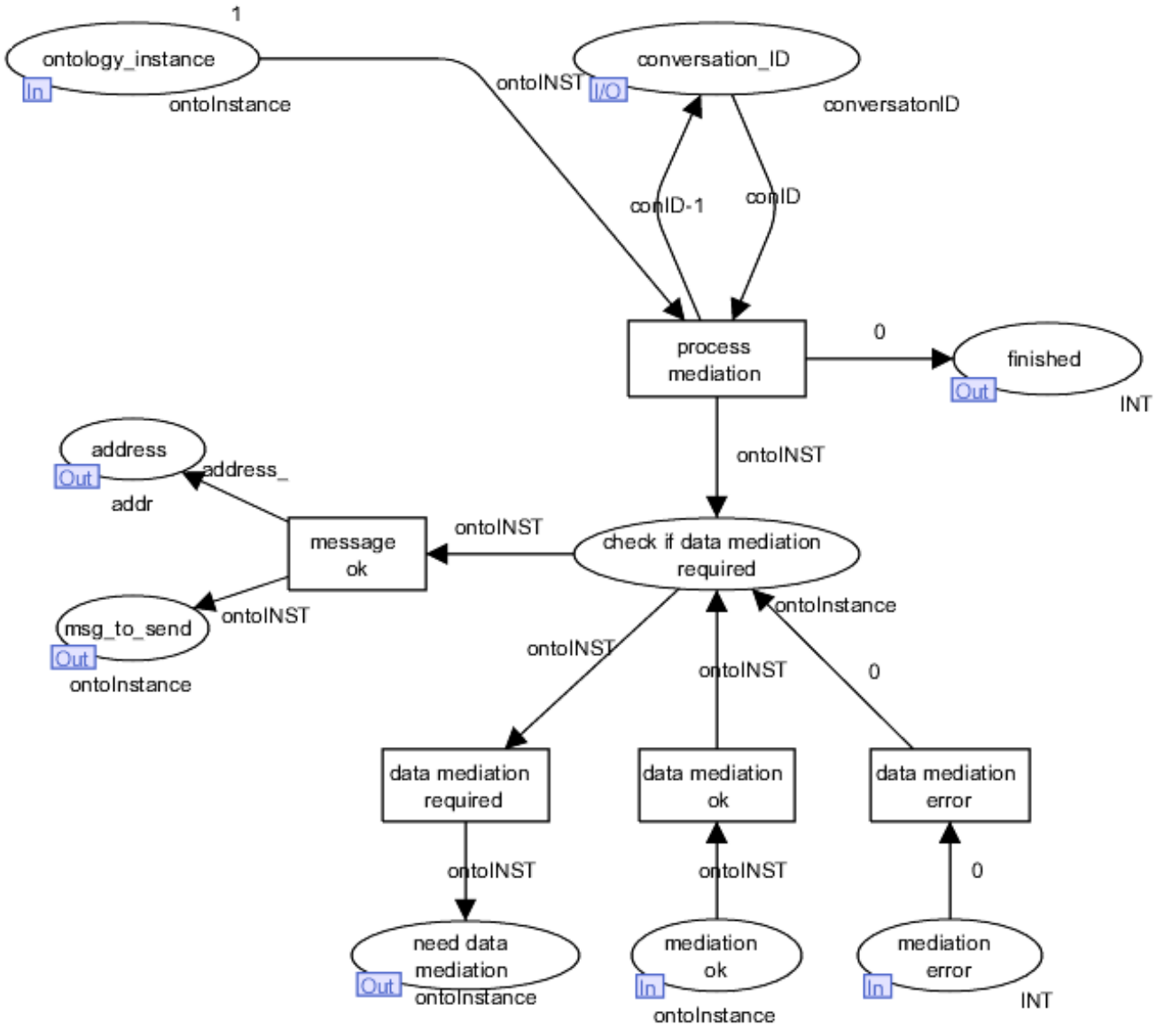


Figure 2. Data Mediation in WSMX

If there is a need for data to be mediated, the source instances are provided to data mediation component which has the role to deliver either the mediated data or an error message. In WSMX the data mediation component is part of more complex interactions, for example the one required by the process mediation:



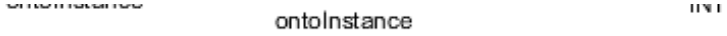


Figure 3. Data Mediator used by the Process Mediator inside WSMX

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 are 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 inputs and guides the user 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. 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 mediation approach, directly influencing the quality, effectiveness and efficiency of the mediation process.

In this section we discuss a way of representing these mappings as well as a methodology for their creation, investigating how the mappings act on the data to be transformed. 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.

3.1. 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 a way that offers maximum reusability as well as great expressivity in order to accommodate a large number of mediation scenarios.

To express our mappings, we chose to use a subset of the mapping language developed in the 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 out 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::='attributeOccurenceCondition('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 abstraction 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 into atomic entities that are to be discovered and stored as such in a persistent storage during design-time and retrieved and composed in rules during run-time. These atomic entities are classified into 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')'`. We consider in the next sections for the sake of simplicity the **classExpr** and **attributeExpr** as being simple *classId*s and *attributesId*s (in the future versions of this deliverable the possibility of having more complex expressions for classes and attributes will be considered). Section 3.2 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 3.2.
- **Conversion functions.** The conversion functions describe how the attribute values from the instances to be mediated are actually transformed (they correspond in the mapping language to logical expressions). The *Conversion functions* will be addressed in future versions of this work.

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

3.2. Mappings Creation

The mapping creation process represents one of the most important phases in a mediator system. It is a design-time process and 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 these mappings we can transform this error-prone and laborious process from a manual task into a semi-automatic one.

In our approach, 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 of interest for this particular task. This step involves different perspectives 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 capturing these 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 use predefined mapping patterns [de Bruijn et al., 2004].

What we propose 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 capture these relationships. Mapping patterns can fill this gap only partially, the step from the semantic relationships to these patterns still remains unspecified.

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 to 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.

3.2.1. 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 contains a subset of the ontology entities (e.g. concepts, attributes, relations, and instances) and the relationships existing between them. Usually the view used for browsing the source ontology (source view) and the view used for browsing the target ontology (target view) are of 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 are a predefined number of roles which 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 to ontology entities. The roles that can be identified in a view are: *Compound Item*, *Primitive Item*, and *Description Item*.
- **Compound Item**. A *Compound Item* has 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 4 presents an abstract representation of a view and the main elements it consists of.

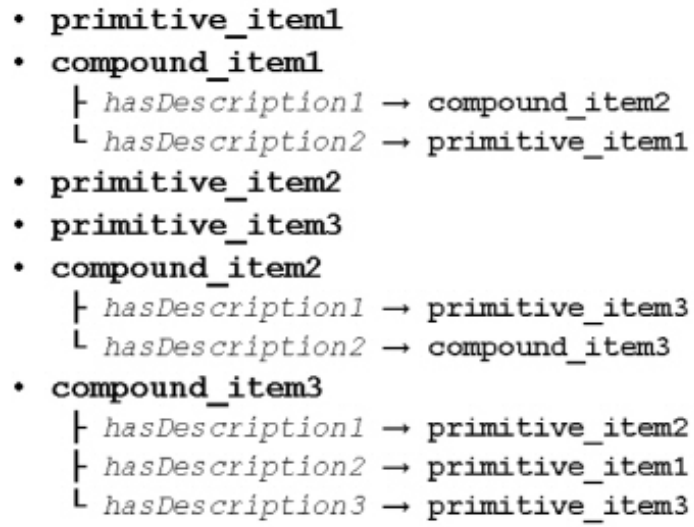


Figure 4. Abstract View

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:

Listing 2. Decomposition algorithm

```
decompose(CollectioncollectionOfItems){
   CollectionresultCollection;
   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(succesorItem))
                  result = result + successorItem;
            }
   }
   returnresult;
}
```

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 to a set of compound items. By applying the algorithm

recursively and updating the corresponding context, the domain expert is guided through the mapping process until all the items from the initial contexts have been 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 target items to which the decomposition process will be applied. 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 to the source and 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*. The decomposition returned a non-empty set of items only for the source 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 make decisions 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 represented by lexical-based algorithms and the second 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 *eligibility factor* (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 items. 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 a 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 than 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.

3.2.2. *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* (e.g. 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 5 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:

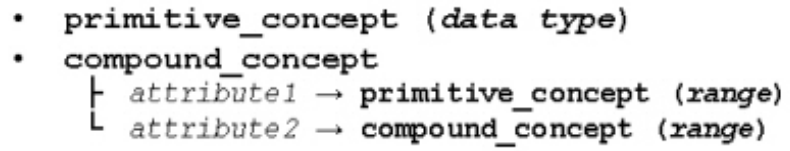


Figure 5. Elements of *PartOf* View

- **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 this type of mapping. Such mappings of this type that might seem initially necessary, are covered by considering a compound concept from the source that owns (or inherits) an attribute pointing to the primitive concept and mapping it with the compound target concept.
- **Compound Concept to Primitive Concept mappings**. This is a situation identical with the above one and the *PartOf* view does not allow these types of mappings either. The rationale 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 type of 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 these 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.

It is worth noting that no conditions are associated to the mappings created by using the PartOf view. For more complex mappings, conditioned by particular characteristics of the source or target items, a different view has to be used.

3.2.3. *InstanceOf* View

During the modelling process a set of instances can be used to properly capture some of the features of the domain (similar to constants in the programming languages). 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 6). 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 item descriptions.

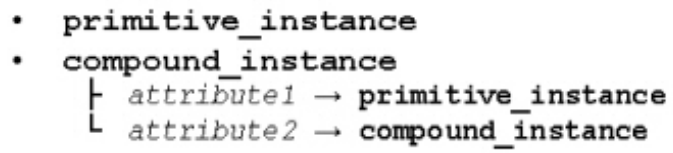


Figure 6. Elements of *InstanceOf* View

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 an *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 for similar reasons to those given for 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 attribute 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 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 the 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.

Using this view we can create mappings that hold only for particular cases of the source and target items. Usually these kinds of mappings are necessary when only a subset of the source and target domains overlap.

3.2.4. *RelatedBy* View

Another interesting view to consider is the one where *attributes* play the roles of items. Each item has two fixed descriptions, one called *hasDomain* and the other called *hasRange*, having as successor the domain and the range of the attribute, respectively. Because of these two fixed descriptions, all the items are *compound items*. In addition, applying the decomposition algorithm to the descriptions and their successors will trigger a change of the view type. As the successors are concepts, by applying decomposition the *RelatedBy* view (see Figure 7) will switch to the *PartOf* view.

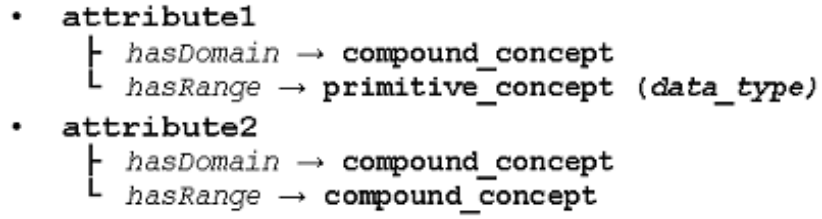


Figure 7. Elements of RelatedBy View

We have two interesting types of mappings that can be done using this view: mappings between two *RelatedBy* views, and mappings between a *RelatedBy* view and a *PartOf* view (for the source and target respectively). For the first case we simply have:

- **(Compound) Attribute to (Compound) Attribute Mapping**. This is a classical mapping between two attributes, which will be followed after applying the decomposition algorithm by mappings between their domains and ranges. Such mappings can be done using the *PartOf* view as well, with only one difference: when created using the *RelatedBy* view, inverse attributes can be mapped, simply by mapping *hasDomain* from the source with the *hasRange* description from the target and *hasRange* from the source with *hasDomain* from the target. One has to keep in mind that the decomposition will trigger a view switching (from *RelatedBy* to *PartOf* view) so these mappings are affected by the restriction of the *PartOf* view (i.e. no mappings between primitive and compound concepts or vice versa are allowed). This type of mapping will generate an *attributeMapping* and a pair of *classMappings* statements in mapping language.

When mapping between RelatedBy and PartOf view (and vice versa) we can have the following types of mappings:

- **(Compound) Attribute** (from *RelatedBy*) **to Compound Concept** (from *PartOf*)**.** This mapping can be followed by any mappings between *hasDomain* and *hasRange* descriptions and descriptions of the compound concept from the target (i.e. attributes) as long as the mappings between successors conform to the restrictions on mappings in the *PartOf* view. A *classAttribute* mapping will be generated, followed by one or more pairs of *classAttributeMapping* and *classMapping*.
- **Compound Concept** (from *PartOf*) **to (Compound) Attribute** (from *RelatedBy*)**.** This is a similar type of mapping as the one presented above. .

In the next section we exemplify how a small ontology fragment is captured through the three types of views presented above.

3.2.5 Examples

Let’s consider the following fragment of ontology (described using WSML v0.1):

Listing 3. WSML Ontology Fragment

```
concept person
      name ofType xsd:string
      age ofType xsd:integer
      hasGender ofType gender
      hasChild ofType person
      marriedTo ofType person

concept gender
      value ofType xsd:string

instance male memberOf gender
      value hasValue ″male″

instance female memberOf gender
     value hasValue ″female″
```

The *PartOf*, *InstanceOf*, and *RelatedBy* corresponding to this fragment are shown in Figure 8.

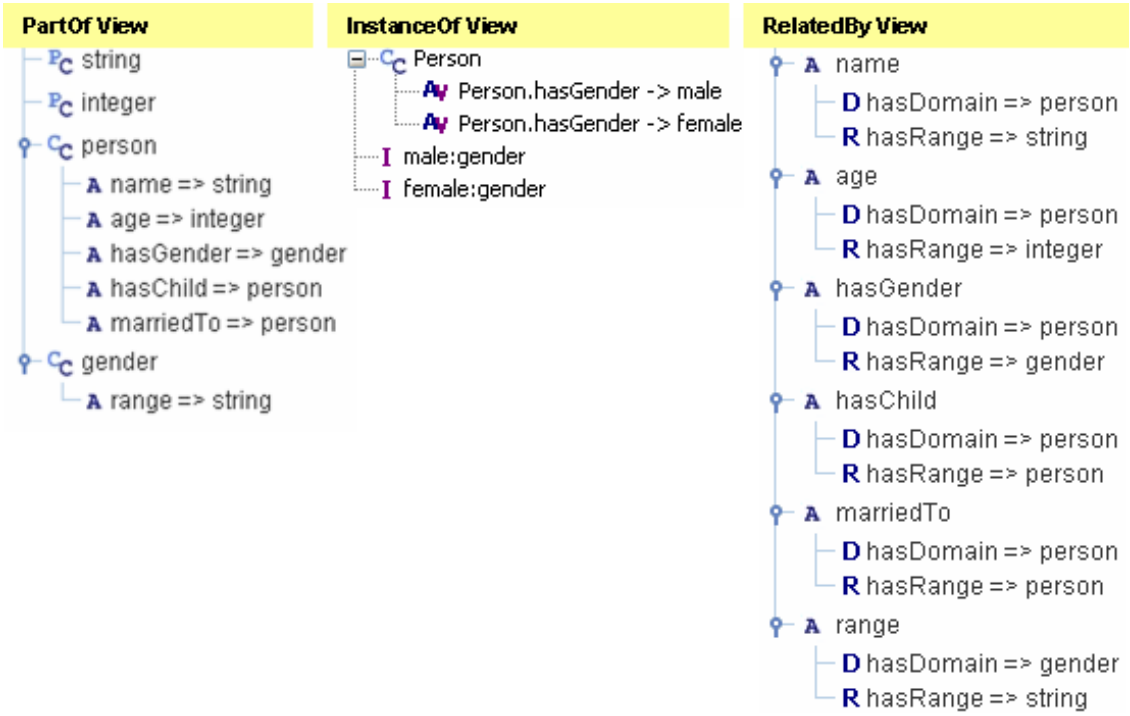


Figure 8. Example of PartOf, InstanceOf, RelatedBy Views

3.3. Mapping Rules Creation

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: it simplifies the mappings maintenance and management, it offers flexibility in choosing the appropriate grounding language in conformance with each specific application requirement 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 to 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 ontologies used 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.

3.4. 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.

4. Flora-2 - A Reasoner for WSMX Data Mediation

As described in Section 3.3, mapping rules are generated based on the mappings created during the design-time phase and stored in persistent storage. These mapping rules are language dependent and their task is to enable the appropriate reasoner to execute the actual instance transformations. The language chosen to express the mapping rules is usually the language in which the input ontologies are expressed. As our scope is WSMO, the ontology language we assume our ontology would be expressed in (in most cases) is WSML [WSML, 2005]. Unfortunately at the time of writing this document no complete implementation for 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 becomes available. As discussed in [WSML Reasoner, 2004] Flora-2 [Flora-2] is a good candidate for reasoning with different WSML variants.

Choosing Flora-2 as a reasoner for WSMX Data Mediator has the followings consequences: transformations from WSML to Flora-2 and grounding from the abstract mapping language to Flora-2 are required. The next two subsections, 4.1 and 4.2, describe in more detail these aspects, respectively.

4.1. From WSML to Flora-2

The WSMX Data Mediation component requires transformations between WSML and Flora-2 only for the runtime phase; during design-time WSML ontologies are loaded and the mappings are expressed in an abstract way. But during run-time the incoming instances (data to be mediated) are expressed in WSML and they have to be processed by the Flora-2 reasoner. To make this possible, these instances have to be transformed first into Flora-2 instances and only then loaded in the Flora-2 environment (Listing 4. shows the example of an instance expressed both in WSML and Flora-2).

Listing 4. Example of an instance expressed in WSML (left) and in Flora-2 (right)

```
instance my_ticket memberOf ticket
      type hasValue "flight"
      lastName hasValue "Mocan"
      firstName hasValue "Adrian"
      ssuing_terms hasValue my_terms
      departure_time hasValue my_departure_time
      departure_date hasValue my_departure_date
      departure_code hasValue "station0341"
      departure_city hasValue "Galway"
      arrival_time hasValue my_arrival_time
      arrival_date hasValue my_arrival_date
      arrival_code hasValue "station0102"
      arrival_city hasValue "Dublin"
```

```
my_ticket:ticket[
      'ticket.type' -> 'flight',
      'ticket.lastName' -> 'Mocan',
      'ticket.firstName' -> 'Adrian',
      'ticket.issuing_terms' -> my_terms,
      'ticket.departure_time' ->
my_departure_time,
      'ticket.departure_date' ->
my_departure_date,
      'ticket.departure_code' -> 'station0341',
      'ticket.departure_city' -> 'Galway',
      'ticket.arrival_time' -> my_arrival_time,
      'ticket.arrival_date' -> my_arrival_date,
      'ticket.arrival_code' -> 'station0102',
      'ticket.arrival_city' -> 'Dublin'
].
```

In addition, some conditions associated to the mappings might refer to schema characteristics, in which case elements of the WSML ontology schema have to be transformed into Flora2 as well and made available to the reasoner. After the mediation process ends, the mediated data (target ontology instances) have to come out from the reasoner as WSML instances. These final transformations can be done by special predicates, directly in Flora2, and the results retrieved directly in WSML by querying the system.

As a consequence, we can say that our main point of interest in WSML to Flora-2 transformation is WSML conceptual syntax transformations, and in particular transformations for instances, concepts and attributes. Appendix A describes the transformation functions used to translate (partially) WSML conceptual syntax to Flora-2.

4.2. 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 transformations functions for the mapping language statements to Flora-2, having the following form:

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

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 to the mapping language statements, Flora-2 rules are obtained. The strings between single quotation signs appear as such in the mapping language or in Flora-2 (depending on which side of the → sign they 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 like

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

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 A (i.e.

transformations function between WSML and Flora-2). In order to avoid confusions the grounding function is called *T*, while for the transformation functions between WSML and Flora-2 the *t* symbol is used.

Listing 5. Grounding general elemnts of the language

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

T(typedLiteral) →t(typedLiteral)

T(plainLiteral)→t(plainLiteral)

T(logicalExpression) → t(logicalExpression)
T(logicalExpresion, 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 need to be syntactically substituted during the translation process (e.g. the variable *X* inside of the logical expression needs to be substituted by *Y)*.

Listing 6. 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(attributeMapping_n, logicalExpression_q)'.'

1<=p<=m
1<=q_1<=q
1<=n_1<=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 to the source and the others to 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 is 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 7. Grounding the class expressions

```
T('and('classExpr_1 ... classExpr_n')', X) →
     T(classExpr_1, X) ',' ... ',' T(classExpr_n, X)

T('or('classExpr_1 ... classExpr_n')', X) →
     '('T(classExpr_1, X) ';' ... ';' T(classExpr_n, X)')'

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

T('join('classExpr_1 ... classExpr_n
     logicalExpression_1...logicalExpression_q')', f(X_2, ... ,X_n)) →
        T(classExpr_1, f(X_2, ... ,X_n))

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) 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 8 have to be considered in addition to the rules generated by the transformations in Listing 7.

Listing 8. Auxiliary transformations for class expressions containing *join*

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

Listing 9 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 9. 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'attributeExpr1attributeExpr2
     attributeCondition 1...attributeCondition n')')→
          T(attributeExpr2, mediated(X), mediated(Y))':-'
              mediated(Y)':-' ',' T(attributeExpr1, X, Y)','
              T(attributeCondition 1, mediated(X), attributeExpr2)
              ',' ...
              T(attributeCondition p, mediated(X), attributeExpr2)','
              T(attributeCondition p+1, X, attributeExpr1)
              ',' ...
              T(attributeCondition n, X, attributeExpr1)'.'
          T(attributeExpr2, mediated(X), Y)':-'
              '\+'mediated(Y)':-' ',' T(attributeExpr1, X, Y)','
              T(attributeCondition 1, mediated(X), attributeExpr2)
              ',' ...
              T(attributeCondition p, mediated(X), attributeExpr2)','
              T(attributeCondition p+1, X, attributeExpr1)
              ',' ...
              T(attributeCondition n, X, attributeExpr1)'.'

T('attributeMapping(one-way'attributeExpr1attributeExpr2
     attributeCondition 1...attributeCondition n')',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 to the source attributes or to the target attributes.

The transformations for the attribute mappings (Listing 10) 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 10. 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)
```

The *inverse*, *symmetric*, *reflexive* and *transitive* attributes are not exploited for now in our approach, and as a consequence they are ignored by the grounding mechanism.

The class conditions transformation function in Listing 11 has an additional parameter representing the instance that owns the attribute which the condition is applied to.

Listing 11. 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('attributeOccurenceCondition('attributeID')', X) →
     X'['T(attributeID) '=>' _']'
```

In Listing 12 the attribute conditions are transformed by using a transformation function with two additional parameters: the first represents the owner of the attribute to which value the conditionis applied, and the second the attribute itself. Please note that the attribute we mentioned above might be an attribute expression as well.

Listing 12. 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)
```

5. Prototype Implementation

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. The first module represents what we called the design- time component while the second two modules represent the run-time component. Figure 9 shows how these components interact.

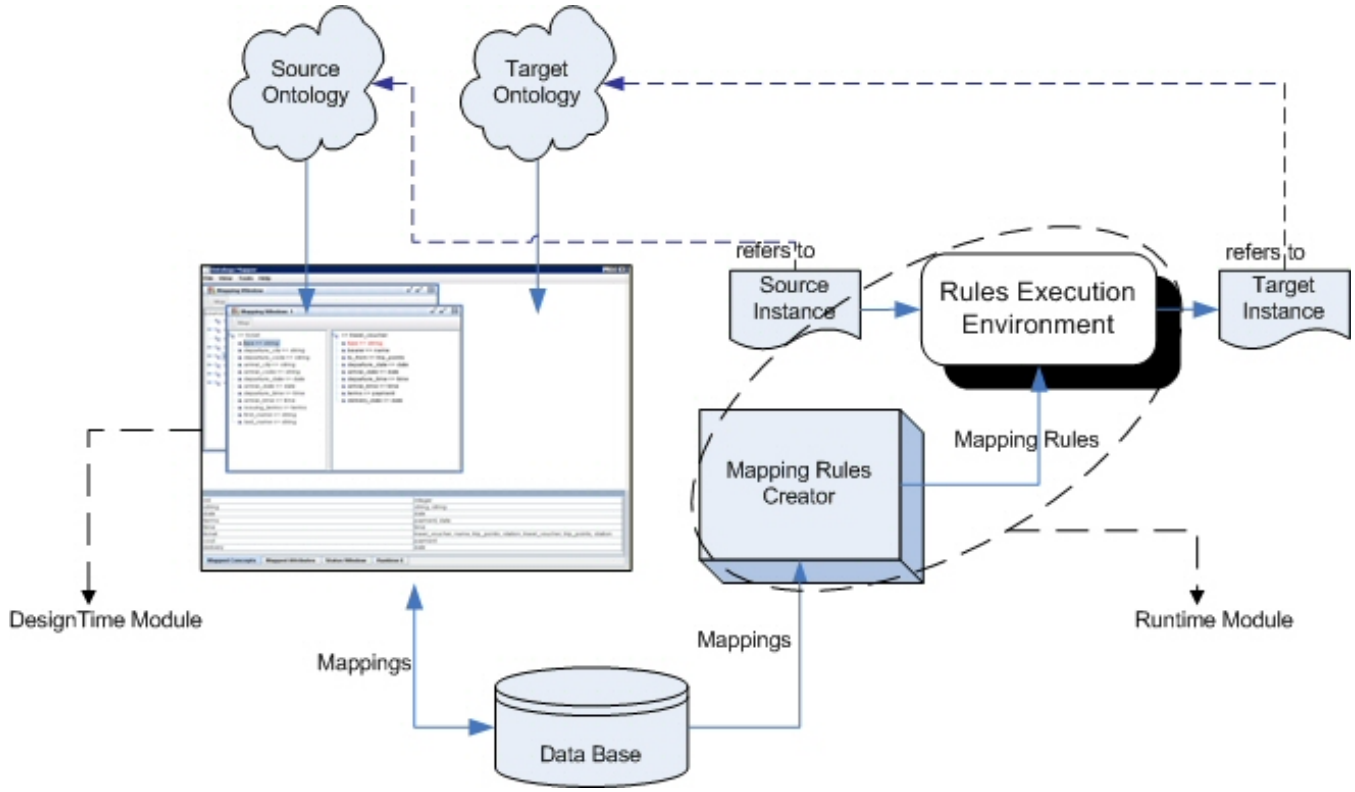


Figure 9. Overview of the Data Mediation Module

5.1. Design Time Component

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. Related work in this direction can be also find in [Maedche et al., 2002] and [Noy & Musen, 2000].

The graphical interface is built using Java 1.5 (see http://java.sun.com for more information) and the default ontology language supported is WSML. For future versions we will consider the creation of various wrappers that could force ontologies expressed in other languages to WSMO conceptual model for ontologies (a simplified one for Flora-2 already exists). The object model behind the mapping tool is fully compatible with WSMO API and WSMO4J (and this applies to the runtime component as well). WSMO API provides a set of interface for manipulating WSMO entities, including ontologies. WSMO4J provides an implementation of these interfaces and a parser for WSML (for more details see https://sourceforge.net/projects/wsmo4j). The mapping tool was integrated as a plug-in in the Web Service Modelling Toolkit (WSMT) [WSMT, 2005], which also offers as plug-ins a WSML editor and a WSMX invoker (see Figure 10). The editor can be used by the domain expert to operate on the ontologies they are mapping and the framework offers basic synchronization functions like reloading the ontologies into the mapping tool or invalidating mapping rules affected by changes in the ontologies. The WSML invoker can be used to test mappings with run-time components deployed as Web Services, directly from the mapping tool (more details about this are provided in the next section).

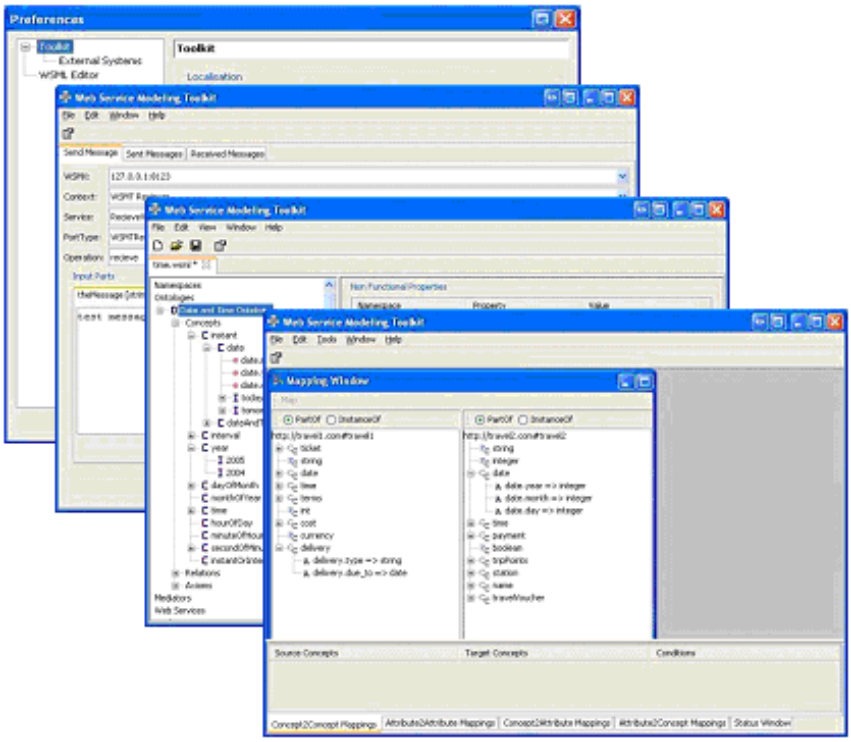


Figure 10. Screenshots of WSMT Plug-ins (from back to front: the Preferences Panel, WSMX Invoker, WSML Editor and Mapping Tool)

As the mapping tool and WSMT are two of the surrounding efforts around WSMO and WSMX, they are part of the WSMX open source project (see https://sourceforge.net/projects/wsmx) - Windows and Linux installers for WSMT can be found on SourceForge (downloads available at: https://sourceforge.net/project/showfiles.php?group_id=113321&package_id=147251).

The 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.

5.2. Run-time component

The Mapping Rules Creator implements the grounding mechanism described in Section 4.2 while the Rule Execution Environment has the role of executing the mapping rules against the incoming source ontology instances. The Run-time component is designed to be part of the WSMX architecture [WSMX Architecture, 2005], and as a consequence it offers well-defined interfaces explaining how it can be invoked (how it can be used is described in Section 2.3). These interfaces are presented in Table 1.

Table 1. Interfaces of the Run-time Mediator

Method Summary	
`Map<Identifiable, List<Identifiable>>`	`mediate(Ontology sourceOntology, Ontology targetOntology, Set<Identifiable> data)` Transforms a set of source ontology instances into instances of the target ontology.
`List<Identifiable>`	`mediate(Ontology sourceOntology, Ontology targetOntology, Identifiable data)` Transforms a given source ontology instance into instances of the target ontology.

StringBuffer	mediate(Ontology sourceOntology, Ontology targetOntology, StringBuffer payload) Transforms source ontology instances into instances of the target ontology. The payload represents a WSML document containing the instances to be mediated. It will be parsed and after the mediation takes places a new WSML document is created containing the target instances.

As one can notice, the first two methods refer to the ontological entities in terms of WSMO API objects, while the last method manipulates WSML documents embedded in *StringBuffer* objects. We recommend the usage of the first two methods because the inputs and outputs have a very precise meaning (they are used in WSMX architecture). One can use the third method when no WSMO API and parsing support is available (suited for tests of the mapping rules existing in the storage).

Outside the WSMX architecture the Run-time component can be used in the following ways:

- As a standalone application able to connect to the provided mapping storage and to perform mediation of instances provided as WSML documents. It offers a small graphical interface where the user can set the source and the target ontology, provide the data to be mediated and retrieve the mediated data.

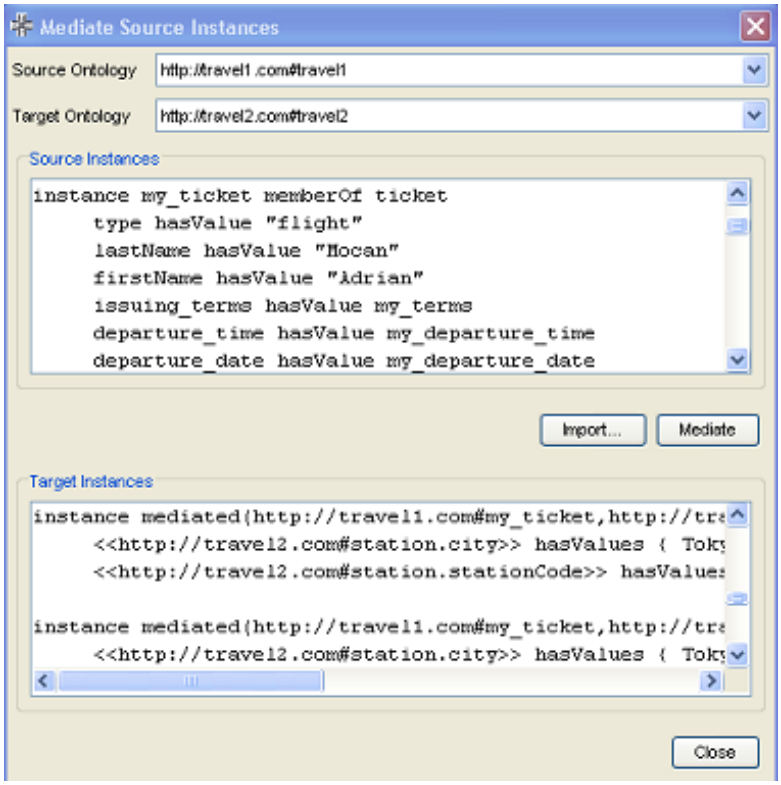


Figure 11. The Graphical Interface of the Stand-alone Run-time Mediator

- As a deployed Web Service (its WSDL file can be found at: http://140.203.154.164:8080/wsmx/services/RuntimeMediator?wsdl) that can be invoked with the source and target ontology IDs and data to be mediated and that returns the mediated data. The mapping tool can directly invoke such Web Services by using the WSMT Invoker. Another option would be to provide a web page as a front end for the run-time mediator.

6. Conclusions and Further Directions

This deliverable provides solutions for ontology-to-ontology mediation problems, with direct application in instance transformation. Our intentions are to provide conceptual foundations for a semi-automatic approach to ontology-to-ontology mediation, together with a software implementation as tool support and as a test bed and a proof of their validity. The software implementation is integrated into the Web Service Modelling Toolkit (the design-time-related components) and into the Web Service Execution Environment (the run-time components) as a default implementation of the mediation component.

In our future work, the focus will be on improving the solutions proposed for ontology mediation, investigating other types of views that could be relevant to our approach, enhancing the mappings and the mapping rules with transformation functions, and creating better heuristics for computing suggestions. In addition, we intend to provide evaluations of both the mapping tool (e.g. in terms of suggestion algorithms accuracy) and run-time tool (e. g. in terms of execution time and accuracy).

References

[Chalupsky, 2000] H. Chalupsky: OntoMorph: A Translation System for Symbolic Knowledge. In *Proceedings of 7th InternationalConference 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.

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

[Fensel & Bussler, 2002] D. Fensel, C. Bussler: The Web Service Modeling Framework (WSMF). In *White Paper and Internal Report Vrije Unversiteit 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. F. 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 onEngineering 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, Seatle, WA, 2001.

[Wiederhold, 1994] G. Wiederhold: Interoperation, Mediation, and Ontologies, In *Proceedings International Symphosium 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, 2005] D. Roman, U. Keller, H. Lausen (eds.): Web Service Modeling Ontology, version 1.2, 2004, available at http://www.wsmo.org/2004/d2/v1.2/

[WSMT, 2005] M. Kerrigan: Web Service Modeling Toolkit (WSMT), WSMX Working Draft, version 0.2, available athttp://www.wsmo.org/TR/d9/d9.1/v0.2/

[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

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(ncname$_1$ ':' ncname$_2$) →
    t(ncname$_2$)

t('_' ncnamechar$_1$ ... ncnamechar$_n$) →
    '_' ncnamechar$_1$ ... ncnamechar$_n$

t([0x0041 ... 0x005A] ncnamechar$_1$ ... ncnamechar$_n$) →
    '"[0x0041 ... 0x005A] ncnamechar$_1$ ... ncnamechar$_n$"'

t([0x0061 ... 0x007A] ncnamechar$_1$ ... ncnamechar$_n$) →
    [0x0061 ... 0x007A] ncnamechar$_1$ ... ncnamechar$_n$

//id - anonymous
t(anonymous) →
    anonymous

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

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

t(number) →
    number

t(string) →
    string

t('true') →
    'true'

t('false') →
    'false'

NonFunctionalProperties

//nfp
t('nfp' attributevalue$_1$ ... attributevalue$_n$ 'endnfp') →
    'nonFunctionalProperties' '->' '_#' '[' *t*(attributevalue$_1$) ',' ... ',' *t*(attributevalue$_n$) ']'

t('nonFunctionalProperties' attributevalue$_1$ ... attributevalue$_n$ 'endNonFunctionalProperties') →
    'nonFunctionalProperties' '->' '_#' '[' *t*(attributevalue$_1$) ',' ... ',' *t*(attributevalue$_n$) ']'

Concepts

//concepts
t('concept' id superconcept nfp attribute$_1$ ... attribute$_n$) →
    t('concept' id superconcept) ',' *t*('concept' id nfp attribute$_1$ ... attribute$_n$)

t('concept' id superconcept nfp) →
    t('concept' id superconcept) ',' *t*('concept' id nfp)

t('concept' id superconcept attribute$_1$ ... attribute$_n$) →
    t('concept' id superconcept) ',' *t*('concept' id attribute$_1$ ... attribute$_n$)

t('concept' id superconcept) →
    t(superconcept, id)

t('concept' id nfp attribute$_1$ ... attribute$_n$) →
    t('concetp id nfp) ',' *t*('concetp id attribute$_1$ ... attribute$_n$)

t('concept' id nfp) →
    t(id) '[' 'nonFunctionalProperties' '->' *t*(nfp) ']'

t('concept' id attribute$_1$ ... attribute$_n$) →
    t(id) '[' *t*(attribute$_1$) ',' ... ','
        t(attribute$_n$) ']'

t('concept' id) →
    t(id)

//superconcept
t('subConceptOf' '{' id ',' id$_1$ ',' ... ',' id$_n$ '}', X) →
    t('subConceptOf' id, X) ',' *t*('subConceptOf' id$_1$, X) ',' ... ',' *t*('subConceptOf' id$_n$, X)

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

Attributes

//attributes
t(id_1 'ofType' id_2) →
    t(id_1) '=>' *t*(id_2)

t(id_1 'ofType' id_2 nfp) →
    t(id_1) '[' 'nonFunctionalProperties' '->' *t*(nfp) ']' '=>' *t*(id_2)

t(id_1 'impliesType' id_2) →
    t(id_1) '=>' *t*(id_2)

t(id_1 'impliesType' id_2 nfp) →
    t(id_1) '[' 'nonFunctionalProperties' '->' *t*(nfp) ']' '=>' *t*(id_2)

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

//attribute values
t(id_1 'hasValue' id_2) →
    t(id_1) '->' *t*(id_2)

t(id 'hasValue' '{' id_0 ',' id_1 ',' ... ',' id_n '}') →
    t(id) '->>' '{' *t*(id_0) ',' *t*(id_1) ',' ... ',' *t*(id_n) '}'

Instances

//instances
t('instance' id memberof nfp $attributevalue_1$ ... $attributvalue_n$) →
    t(memberof, id) '.' *t*('instance' id nfp $attributevalue_1$ ... $attributvalue_n$)

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

t('instance' id memberof $attributevalue_1$ ... $attributvalue_n$) →
    t(memberof, id) '.' *t*('instance' id $attributevalue_1$ ... $attributvalue_n$)

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

t('instance' id nfp $attributevalue_1$ ... $attributvalue_n$) →
    t('instance' id nfp) ',' *t*('instance' id $attributevalue_1$ ... $attributvalue_n$)

t('instance' id nfp) →
    t(id) '[' 'nonFunctionalProperties' '->' *t*(nfp) ']'

t('instance' id $attributevalue_1$ ... $attributvalue_n$) →
    t(id) '[' *t*($attributevalue_1$) ',' ... ','

t(attributevalue$_n$) ']'

t('instance' id) →
t(id)

//memberof
t(memberof, X) →
t('memberOf' idlist, X)

t('memberOf' '{' id ',' id$_1$ ',' ... ',' id$_n$ '}', X) →
t('memberOf' id, X) '.' *t*('memberOf' id$_1$, X) '.' ... '.' *t*('memberOf' id$_n$, X)

t('memeberOf' id, X) →
X ':' *t*(id)

Appendix B. Changelog

2005.07.06

- Applying changes/updates/proof reading comments throughout the whole documents. Ready to be made final.

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