Deliverable

WP5: Service Mediation

D5.2

Mediation Module Specification: Business Data-Level Mediation

Atanas Kiryakov, Christian Drumm, Lina Al-Jadir, Aida Boukottaya, Edward Kilgarriff, Joachim Quantz

Jan 31st, 2005
EXECUTIVE SUMMARY

Data-mediation\(^1\) is required in a (semantic) web services environment in order to align the mismatches of the data schemata and the representations, used by the different parties. The approach taken in DIP assumes that:

- Mediators are crafted manually or semi-automatically design-/setup-time;
- An appropriate mediator is found and used for automatic transformation runtime.

The involvement of a human in the design-time definition of a mediator is necessary to assure the correctness of the fully automatic transformation that is performed during execution.

There are two sorts of mediation that can be singled out: semantic (based on ontology-mapping) and syntactic. While the former is more comprehensive, the latter can prove to be useful for efficient handling of the simple cases, whenever possible.

This deliverable presents the proposed overall mediation approach and outlines the related ontology mapping language and components. Further, it elaborates in greater detail on the specification of the various aspects, including a concrete proposal for generation of a syntactic (XSLT) transformation out of the ontology mappings. The specification provided hereby requires further development in order to allow for straightforward implementations. The latter is due to the early stage of development of a number of related aspects – most notably ontology mapping and transformation.

This deliverable gives a boost to WSMO and WSMX through a concrete specification for data mediation presented hereby. This specification is an important step towards the development of the DIP architecture in a way, which provides a possibility for an open architecture with a high industry impact.

The relevance of D5.2 is primarily focused within WP5, allowing for the implementation of data mediation tools (D5.4). It has a bi-directional dependence on WP6 deliverables D6.2 and D6.3, in terms that on one hand it specifies an important aspects of a number of mediation-related components, and on the other hand it defines the latter in accordance with its role in the architecture and the interfaces of the other components that it depends on. The target audience is mostly made up of the technology providers within the DIP consortium.

\(^1\) Within this document, “data-mediation” is used instead of “data-level mediation”.
The objective of this deliverable has been a specification for a data-mediation component. The data mediation task is hereby considered a translation of messages and related instance data between two parties (e.g. a WS requester and a provider), using different WSML ontologies. Two phases of data-mediation are considered: human-aided design-time mediation specification; and automatic run-time mediation execution in the course of WS usage. The high-level design-time task is to create a WSMO ooMediator, which can mediate between a source and a target ontology. There are two sorts of mediation that can be singled out: semantic (based on ontology-mapping) and syntactic. An overall mediation approach is presented, followed by short descriptions and references to the related ontology mapping language and components. One of the concrete contributions is a proposal for generation of a syntactic (XSLT) transformation out of the ontology mappings.

**Keywords**
- Data-mediation
- ontology mapping
- WSMO
- WSML
- WSMX
- XSLT

**Version Log**

<table>
<thead>
<tr>
<th>Issue Date</th>
<th>Rev No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
<td>Atanas</td>
<td>Initial version of document</td>
</tr>
<tr>
<td>Date</td>
<td>Time</td>
<td>Author</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>20-10-04</td>
<td>04</td>
<td>Christian Drum</td>
<td>• Added section 2.2 (Ontology mapping language)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Added first contend to section 5 (Creation of syntactic transformation rules)</td>
</tr>
<tr>
<td>17-11-04</td>
<td>05</td>
<td>Atanas Kiryakov</td>
<td>Integrated content from other partners</td>
</tr>
<tr>
<td>26-11-04</td>
<td>06</td>
<td>Christian Drum</td>
<td>Major update of section 3.</td>
</tr>
<tr>
<td>29-11-04</td>
<td>07</td>
<td>Atanas Kiryakov</td>
<td>Major update of section 1.</td>
</tr>
<tr>
<td>15-01-05</td>
<td>08</td>
<td>Atanas Kiryakov</td>
<td>Gathering updates from the WP5 partners on all sections</td>
</tr>
<tr>
<td>24-01-05</td>
<td>09</td>
<td>Atanas Kiryakov</td>
<td>Accommodation of the input from the plenary meeting in Innsbruck.</td>
</tr>
<tr>
<td>31-01-05</td>
<td>10</td>
<td>Atanas Kiryakov</td>
<td>Accommodation of the input from the internal reviewers and proof reading.</td>
</tr>
</tbody>
</table>
## Project Consortium Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| National University of Ireland Galway | NUIG | Prof. Dr. Christoph Bussler  
Digital Enterprise Research Institute (DERI)  
National University of Ireland, Galway  
Galway, Ireland  
Email: chris.bussler@deri.org  
Tel: +353 91 512460 |
| Fundacion De La Innovacion.Bankinter | Bankinter | Monica Martinez Montes  
Fundacion de la Innovation. BankInter  
Paseo Castellana, 29  
28046 Madrid, Spain  
Email: mmmnez@bankinter.es  
Tel: 916234238 |
| Berlecon Research GmbH | Berlecon | Dr. Thorsten Wichmann  
Berlecon Research GmbH  
Oranienburger Str. 32  
10117 Berlin, Germany  
Email: tw@berlecon.de  
Tel: +49 30 2852960 |
| British Telecommunications Plc. | BT | Dr John Davies  
BT Exact (Onion Floor 5 pp12)  
Adastral Park Martlesham  
Ipswich IP5 3RE, United Kingdom  
Email: john.nj.davies@bt.com  
Tel: +44 1473 609583 |
| Swiss Federal Institute of Technology, Lausanne | EPFL | Prof. Karl Aberer  
Distributed Information Systems Laboratory  
École Polytechnique Fédérale de Lausanne  
Bât. PSE-A  
1015 Lausanne, Switzerland  
Email: Karl.Aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council | Essex | Mary Rowiatt,  
Essex County Council  
PO Box 11, County Hall, Duke Street  
Chelmsford, Essex, CM1 1LX  
United Kingdom.  
Email: maryr@essexcc.gov.uk  
Tel: +44 (0)1245 436524 |
| Forschungszentrum Informatik | FZI | Andreas Abecker  
Forschungszentrum Informatik  
Haid-und-Neu Strasse 10-14  
76131 Karlsruhe, Germany  
Email: abecker@fzi.de  
Tel: +49 721 9654 0 |
<p>| Institut für Informatik, Leopold-Franzens | UIBK | Prof. Dieter Fensel |</p>
<table>
<thead>
<tr>
<th>Mediation module specification: business data-level mediation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Universität Innsbruck</strong></td>
</tr>
<tr>
<td><strong>ILOG SA</strong></td>
</tr>
<tr>
<td><strong>inubit AG</strong></td>
</tr>
<tr>
<td><strong>Intelligent Software Components, S.A.</strong></td>
</tr>
<tr>
<td><strong>The Open University</strong></td>
</tr>
<tr>
<td><strong>SAP AG</strong></td>
</tr>
<tr>
<td><strong>Sirma Al Ltd.</strong></td>
</tr>
<tr>
<td><strong>Tiscali Österreich GmbH</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Unicorn</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Jerusalem</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Vrije Universiteit Brussel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VUB</td>
</tr>
<tr>
<td></td>
<td>Carlo Wouters</td>
</tr>
<tr>
<td></td>
<td>Starlab- VUB</td>
</tr>
<tr>
<td></td>
<td>Vrije Universiteit Brussel</td>
</tr>
<tr>
<td></td>
<td>Pleinlaan 2, G-10</td>
</tr>
<tr>
<td></td>
<td>1050 Brussel ,Belgium</td>
</tr>
<tr>
<td></td>
<td>Email: <a href="mailto:carlo.wouters@vub.ac.be">carlo.wouters@vub.ac.be</a></td>
</tr>
<tr>
<td></td>
<td>Tel.: +32 (0) 2 629 3719</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

**EXECUTIVE SUMMARY** .................................................................................................................. III  
**TABLE OF CONTENTS** ................................................................................................................. IX  
**1 INTRODUCTION** ......................................................................................................................... 1  
  1.1 Definition of scope ......................................................................................................................... 2  
  1.2 Format Adapters ............................................................................................................................. 3  
  1.3 Ontology Integration ....................................................................................................................... 5  
  1.4 Ontology Language ......................................................................................................................... 6  
  1.5 Ontology Design Recommendations ............................................................................................. 6  
**2 MEDIATION APPROACH** ............................................................................................................. 8  
  2.1 Component Specification .................................................................................................................. 10  
  2.2 Mediator Specification .................................................................................................................... 11  
**3 ONTOLOGY MAPPING LANGUAGE** ............................................................................................ 13  
  3.1 Ontology Mapping Patterns .......................................................................................................... 13  
  3.2 Syntax of the Mapping Language .................................................................................................. 15  
  3.3 Evolution of the Mapping Language ............................................................................................ 15  
**4 ONTOLOGY MAPPING-RELATED INTERFACES** .......................................................................... 17  
  4.1 Ontology Mapping Data Model ..................................................................................................... 17  
  4.2 Pattern and Mapping Store Interfaces ............................................................................................ 18  
  4.3 Instance Transformation Interface ................................................................................................ 19  
  4.4 Instance Unification Interface ...................................................................................................... 19  
**5 CREATION OF SYNTACTIC TRANSFORMATION RULES** ............................................................ 20  
  5.1 Creation of syntactic transformations during design-time ............................................................. 20  
  5.1.1 Language-primitive translations .............................................................................................. 21  
  5.2 Creation of syntactic transformations during run-time .................................................................. 26  
**6 CONCLUSION** .............................................................................................................................. 27  
**REFERENCES** ................................................................................................................................ 28  
**APPENDIX A – EXAMPLE OF TRANSLATION OF ONTOLOGY MAPPING TO XSLT** .......... 30  
  Example Mapping ............................................................................................................................... 30  
  XSLT Translation ............................................................................................................................... 30  
**APPENDIX B – ONTOLOGY MEDIATION COMPONENT APIs** ....................................................... 32  
  StoreFactory ....................................................................................................................................... 32
Mediation module specification: business data-level mediation

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatternStore</td>
<td>32</td>
</tr>
<tr>
<td>MappingStore</td>
<td>33</td>
</tr>
<tr>
<td>InstanceTransformation</td>
<td>33</td>
</tr>
<tr>
<td>InstanceUnification</td>
<td>33</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

Data-mediation relating to web services (WS) is discussed in a number of documents in the scope of DIP, WSMO, and WSMX. Related work on ontology mediation is carried out in the scope of the SEKT project as well. Here follows a list of the most relevant documents, taken as an input for this specification:

- the requirement analysis from section 3.2 of deliverable D5.1 of DIP, [6];
- the pre-study of Michael Stollberg on mediation, [23].
- deliverable D6.1 of DIP, [2], providing architectural requirements;
- deliverable D6.2 of DIP, [14], presenting the overall DIP architecture;
- deliverable D13.3 of WSMX, [18], which provides an extended discussion on the overall mediation landscape;
- deliverable D4.3.1 of SEKT, [8], which defines an ontology mapping language;
- Annex 1 of deliverable D4.5.1 of SEKT, [4], which defines the interfaces for ontology mappings and pattern stores.

The data mediation task is hereby considered a translation of messages and related instance data between two parties (e.g. a WS requester and a provider), using different WSML, [9], ontologies. In the general case, a data-mediation is expected to have two modalities: (i) human-aided design-time mediation specification, and (ii) automatic run-time mediation execution in the course of WS usage. The first phase should guarantee a 100% correct schema-/ontology-level mapping, which aims at the generation of an efficient and correct – as far as possible – run-time mediation procedure.

The high-level design-time task is to create a WSMO ooMediator (see [21]), which can mediate between a source and a target ontology. This mediator is looked up and used run-time for translation. A possible run-time mediation scenario is presented in section 5.2 of [14]: after the discovery phase, the requester “understands” that it has no ontology in common with the provider; thus, in the course of the communication, it makes a mediation request to the execution environment for each message; the latter looks up and invokes a mediator, then returns the translated message to the requester, which can then send it to the provider.

The rest of this section provides a discussion on the scope of the data-mediation problem (most important, makes clear what will not be addressed in this deliverable) and a number of related issues: format adapters, ontology integration, language, and design. Section two describes a concrete mediation approach together with the tools and components required for its realisation. The third and the fourth sections present (and refer to) two SEKT developments which fit in this specification: the ontology mapping language and the interfaces for an ontology mediation component. An approach for syntactic transformation, based on ontology-mapping is described in the fifth section. Section six is dedicated to a summary and a conclusion of the deliverable. Finally, there are two appendices: Appendix A provides an example for syntactic transformation; Appendix B presents a JavaDoc-style specification of the interfaces of the ontology mediation component.
1.1 Definition of scope

In the scope of DIP, data-mediation is considered the message and/or instance transformation from one WSML ontology to another, using an ooMediator. There are a number of issues, which are, or could be, related to data-mediation, but which still remain outside the scope of this specification:

- The concrete interaction schema for usage of the mediators. Depending on the execution environment configuration, data mediation could be performed in a number of different manners. For instance, in contrast to the scenario mentioned above, the mediation could be made transparent to the requester through a brokerage-based setup. Under specific circumstances, it could also be pushed to the WS provider side, which could enable more advanced patterns, involving instance-level unification (see [4]);

- Any format transformations to and from WSML are to be handled by the adapters, as discussed in sections 4.1, 4.4, and 5.2 of [14]. We provide some discussion on the behaviour of the adapters in section 1.2;

- The integration of non-ontological data-sources (e.g. databases) with ontologies and an ontology management infrastructure (repositories, reasoners, editors, etc) remains outside the scope of data-mediation. It is considered a data-integration problem that is addressed to some extent in the ontology management workpackage WP2 – a conceptual model is presented in [15], while the implementation will be part of the ORDI framework, D2.3. A more general overview of the state-of-the-art industrial data-integration platforms and systems can be found in [1];

- The mediation between a goal and a WS may also include data-mediation. Actually, as long as all the requirements of the requester can be specified as a goal, then most of the WS-related actions, including the required data transformation, can be regarded as goal satisfaction or goal-to-something mediation. However, for instance, the goal to capability matching bears discovery and match-making logic, which sorts it beyond the scope of this specification. Still, it is obvious that within concrete execution environments the data-mediation functionality could be amalgamated with goal-to-goal and WS-to-goal mediation;

- An important mediation case is the alignment of the granularity of the messages or the structure of the parameters. For instance, it may prove necessary that the values of two parameters are somehow combined into a single one. This task is beyond the scope of this specification, as far as it requires a web-service-description awareness, i.e. it is not a pure data-mediation problem. Still, in some environments and implementations, this could become an integrated feature of a data-mediation module;

- The import, transformation, and adaptation of WS and SWS descriptions (e.g. in WSDL or OWL-S) to WSML is not a data-mediation problem;

- The tools and methodologies for establishment or development of ontology mappings are not the subject of this specification. Ontology mapping and merging is a broad and widely studied area (see [7] for a recent state-of-the-art study). In the context of the SDK cluster, considerable efforts have been
dedicated to it within the SEKT ([7], [4]) and KnowledgeWeb ([13]) projects. Further, within DIP, a framework for mapping of ontologies is developed in D1.5, while ontology mapping and merging tools will be developed in D2.6. The variety of the ontology integration approaches is discussed in section 1.3. The ontology mapping language is presented in section 3. Ontology mapping-related interfaces and components are referred to in section 4.

Finally, the co-ordination/correspondence/integration between the data-mediation and process-mediation components is hard to be resolved at this stage, because at present the latter is still underspecified. Still, the specification for data-mediation can provide some important requirements and prescriptions for the process-mediation specification (D5.3).

1.2 Format Adapters

As introduced in the DIP Architecture ([14] section 4.4), adapters enable systems that do not represent data using WSML to interact with DIP. Adapters are bi-directional facilities. In the most obvious case of WSDL web services exchanging XML messages, the adapter makes sure that the data, sent from a web service, corresponding to the XML schema specified in the WSDL interface, gets “lifted” to the WSML representation expected from the rest of the components. In the opposite direction, a WSML message sent to a non-WSML-speaking system must be “lowered” to the corresponding proprietary XML format.

The adapters only take care of the syntactic transformation, i.e. they transform XML into a proprietary WSML representation. They should typically handle representation mismatches only, not structural or semantic ones, which means that:

- those can be fully automatic;
- they are used before and/or after the essential data mediation, as shown on Figure 1.

![Figure 1. Business data mediation on the ontology level (from [6])](image-url)
It should be taken into account that the adaptation also has a schema-level aspect. The efficient transformation of instances (elements, messages, parameter values, etc), e.g. from XML or OWL into WSML, requires that the corresponding schema also gets “lifted” into a WSML ontology beforehand. In many cases this transformation can as well be handled automatically.

Let us elaborate on an oversimplified scenario, in which there is a single requester that works with an XML schema X1, and a single provider, which provides a web service with a WSDL specification that refers to an XML schema X2. Suppose that there is a universal bi-directional XML-to-WSML adapter, which can perform the following actions, presented as functional calls:

- **liftSchema(xml_schema)** – generate a WSML ontology out of an XSD schema. This method can be used design-time to create an ontology O1 out of X1 and an ontology O2 out of X2;
- **liftMessage(xml_message)** – generate a WSML instance out of an XML message. To perform this, the adapter finds (or generates on demand) the ontology corresponding to the scheme of the message (e.g. ontology O1 for schema X1) and generates the instance with respect to this ontology.
- **lowerMessage(instance)** – generates an XML message out of a WSML instance. To perform this, the adapter determines the ontology used and then finds the corresponding XML schema; next the message is generated with respect to this schema.

Note that no schema lowering is required – there is no interest in the creation of new non-WSML schemata/ontologies. The adapter works with the assumptions that:

- it is possible to determine a single ontology, OL, used to define the instance;
- OL is a result of a lifting of a schema, so, it is straightforward to determine its corresponding schema.

In parallel to its high level of idealization, the above mentioned scenario also bears a number of beautiful traits. On one hand, the schema lifting is performed automatically, design-time or on demand. Further, there is a single component, which is responsible for the schema lifting and the instance lifting and lowering, which ensures the correctness of the instance-level transformations.

A possible implementation approach for such an adapter is the development of two sorts of XSLT scripts used during the schema lifting, respectively for:

- generation of a WSML ontology from a XSD schema;
- generation of XSLT scripts for instance lifting and lowering, based on the XSD and the ontology, generated with the “true” schema lifting XSLTs.

The adapters can be realized in accordance with different strategies and architectures. The above mentioned scenario, including the functionality of the adapter put forward there, is a tentative one; it is meant to serve as a reference to a possible specification and implementation approach. However, it should be considered that the correctness of instance lifting and lowering with respect to the particular ontologies and schemata should be ensured in any event.
It should also be considered that there could be multiple (alternative or complementary) adapters, corresponding to one and the same ontology. In this case, the management of such a situation depends on the implementation of the execution environment.

To finalize the discussion on adapters, we present a couple of references to related work, which can facilitate their definition and development. OntoLiFT, [25], was developed in the course of the WonderWeb project; it specifies a tree grammar based approach and provides partial implementation for lifting of relations schemata, XML schemata, and UML into RDFS. A mechanism for interpretation of XML schemata with respect to the RDF model was developed in the course of the SWWS project and reported in, [5]; it allows for both lifting and lowering of messages.

1.3 Ontology Integration

The data-mediation approach, adapted hereby, counts on semantic modelling of the different data-sources and formats and on a semantic specification of their correspondences. The latter can be achieved, generally, through ontology integration. As outlined in [23], three major strategies for ontology integration can be distinguished. “Ontology mapping” is the case when a correspondence, in terms of mapping rules, is defined to enable interoperability between two ontologies. The rules are created and maintained independently from the ontologies being mapped. “Ontology alignment” is the scenario when one ontology, say O1, is extended in order to accommodate its alignment with another one, say O2. The result could be either an extension of O1, which is dependent on (i.e. imports) O2, or a new ontology, which imports both O1 and O2 and ontology definitions, which align O1 to O2. “Ontology merging” unites the ontologies of interest into a new one, which comprises all their definitions and semantics and interconnects them. Figure 2 demonstrates the difference between these approaches.

![Ontology Integration Strategies](image)

Figure 2. Ontology Integration Strategies (FROM [23])

One can outline multiple nuances and variants of these cases when multiple (more than two) ontologies are involved. The ontology integration, described above, considers the schema-level mediation. In the case of the alignment and mapping, there is ultimately a single ontology, which allows a straightforward interpretation of data represented with respect to the different ontologies; thus, no instance-level transformation is required. In the case of mapping, the mapping rules should be used for the transformation of
instance data, represented with respect to one of the ontologies, into such that is represented with respect to the other.

In other words, a mapping enables inter-operation between two loosely coupled Ontologies. An ontology mapping M is a (declarative) specification of the semantic overlap between two ontologies O1 and O2. This mapping can be one-way (injective) or two-way (bijective). In an injective mapping we specify how to express terms in O1 using terms from O2 in a way that is not easily invertible. A bijective mapping works both ways, i.e. a term in O1 is expressed using terms of O2 and the other way around. In [10] mappings are represented in an abstract language, and this abstract representation does not commit to any existing mapping language.

The applicability of alignment and merging depends on the control over the ontologies involved and on the modality of their usage and development. Merging can only be applied easily if the parties, using the different ontologies, can afterwards be convinced to use the merged one. In effect, such a “persuasion” means that they should agree not to develop further their own ontologies independently or to do it in a carefully synchronized fashion. The alignment makes less co-ordination assumptions – it is usually, a problem of the aligning party and varies considerably from one case to another. As long as such co-ordination assumptions do not fit well into a general web service environment, mapping is the main integration approach considered hereby.

1.4 Ontology Language

Within the DIP project, WSMO is accepted as a modelling paradigm and the WSML family of specifications and languages ([21],[9]) will be used for formal representation. Thus, WSML is considered as the ontology language for this deliverable. Further, the mapping language specified in [8] is considered as the ontology mapping language. For more details see section 3.

As the exchange format for ontologies we will relay on the WSML XML syntax. This enables us to support, apart from the ontology based data mediation, the usages of syntactic transformations scripts (i.e. XSLT scripts). The goal of this will be a simple integration to the developed concepts into existing infrastructures. For details see section 5.

1.5 Ontology Design Recommendations

The architecture of DIP is not bound to a particular ontology or to knowledge modelling assumptions, apart from WSMO as a WS modelling paradigm. Here we briefly refer to the recommendations of the service description framework (SDF), presented in [3], and we comment on their relations to data-mediation. SDF defines the notion of Business Data Ontologies, addressing the data-models (more or less static), related to a WS description. The notions of a goal ontology and a process ontology are defined together with their roles in the WS description. Naturally, the data-mediation problem is mostly related to business data ontologies.

The recommended approach for modelling of business data is the following. WSMO, on its own, does not provide any assumptions or support for modelling of the world. A concrete Business Data Ontology (BDO), [19], is developed as an upper-level ontology, which defines domain-independent concepts, ranging from general ones (e.g. “Person”) to more business-relevant notions (e.g. “Account”). In fact, BDO has been developed as
an extension of the BULO ontology, which is an early version of the PROTON upper-level ontology, developed within the SEKT project, [24]. BDO inherits the highest-level, non-business definitions from BULO. Further, it is recommended ([3], section 5.1) that domain-specific ontologies be developed as an extension of BDO, or aligned to it, in order to allow for re-use and for easier cross-domain interoperability. Finally, we recommend that the particular organisations develop their data ontologies as an extension of the most specific ontology out of the chain above (PROTON, BDO, DomainX), which suits their needs and requirements best.

In the case of ontologies, which are generated automatically, e.g. through lifting from other data-schemata, we recommend that those should not be directly mapped to other proprietary ontologies, but first be aligned to an appropriate domain or upper-level ontology. This methodology has the potential to enable a more efficient and correct semi-automatic ontology mapping (as compared to a one-to-one mapping between isolated ontologies). Figure 3 presents the dependences (in a right–to-left reading order) in the recommended ontology modelling and mapping scenario.

![Figure 3. Recommended Ontology Design Methodology](image)

It is important to state it clearly once again that only WSMO, and nothing more, is enforced as a modelling assumption. The users (application developers, integrators, etc.) have absolute freedom whether to use an upper-level ontology or not, and also which one to use. The same is also valid for the approach for ontology mapping.
2 Mediation Approach

The Semantic Web Services paradigm employs ontologies for a number of tasks, related to Web Services, including mediation. The overall mediation approach in DIP and WSMO is based on the understanding that data mediation in a heterogeneous business environment is only possible through semantic modelling and alignment. It is backed by the state-of-the-art analysis, [6], which provides sufficient evidence of the potential of the semantically-rich methods and the problems, which the existing industrial data-integration platforms, [1], face even in the scope of a single enterprise.

In order to make semantic mediation possible, in WSMO, it is assumed that all the data is available or exchanged in WSML, which is a combination of an ontology and a WS modelling language (see section 1.4). Whenever necessary, adapters (see section 1.2) are used to transform the data into this format. The data-mediation is performed through ooMediators, handling data transformation between a source ontology and a target one.

As discussed in section 1.3, the essential step during the data-mediation is the establishment of an ontology mapping between the ontologies of the source and target parties. Once ontology mapping is created, it is used for the actual instance-level data transformation. The major specificity of the approach proposed hereby is that we consider the case when, after the ontology mapping, it appears that a syntactic transformation is possible. The key point is that in any case a proper semantic modelling and mapping is performed at the schema/ontology-level, which guarantees the correctness of the judgement regarding the applicability of a purely syntactic transformation at the instance level. This approach has two major beauties:

- it allows for a more efficient (computationally less expensive) syntactic transformation, even when a semantic transformation service is available;
- it allows for usage of syntactic transformation in cases when the semantic one is not applicable.

Let us specify two “universal” mediation services:

- semantic transformation service (SETS);
- syntactic transformation service (SYTS).

Both services take as input a message (or a set of instance data) and a transformation specification, perform the instance data transformation, and return the “translated” message as a result. SETS and SYTS only differ by the type of the specification transformation, which they can interpret.

Let us now assume that the source ontology O1 and the target ontology O2 are already present (together with the appropriate adapters, if such are necessary). The sequence of actions for the creation of an ooMediator looks as follows:

1: Create an ontology mapping OM1_2 between O1 and O2 and store it in a mapping store;
2: Evaluate the applicability of syntactic transformation. In case that such is possible, go through the B variants below;
3: Based on OM1_2, generate a transformation specification out of the proper kind:
3A: A semantic transformation specification, e.g. ontology mapping rules, as an executable form of the mapping;

3B: A syntactic transformation specification, e.g. XSLT scripts;

4: Create an ooMediator as a mediation profile, which specifies source and target ontologies and ontology mapping. According to the output of step 2, further augment the mediator by specifying:

4A: SETS as a mediation service and the transformation procedure as provided at step 3A.

4B: SYTS as a mediation service and the transformation procedure as provided at step 3B.

5: Store/publish the mediator.

It is a matter of implementation of SETS and SYTS whether the step involving the generation of a transformation specification is necessary. A possible scenario is that the mediation service takes as an input the ontology mapping and either interprets it directly or “compiles” it on demand into some executable form. The applicability of this scenario depends on: whether a “compilation” step is necessary; if it is so, whether it can be fully automated; and if it can be, how computationally expensive this task is. We consider the scenario further into the generation of an explicit transformation procedure, as long as the other one can be considered a simplification (the ontology mapping is provided as a transformation specification).

The design-time procedure for the development of an ooMediator for data mediation is presented on Figure 4, including references to the tools involved.

Figure 4. Interaction diagram for the development of ooMediator for data-mediation
There are cases where a semantic transformation service may not be applicable, for instance when it is not available or when the IT infrastructure does not allow its integration for some reason. The approach presented hereby allows that, disregarding the results of the evaluation step, the syntactic procedure can be adapted as a back-up option. Despite that in such cases the correctness may not be guaranteed, in many business scenarios the availability of such an alternative is crucial.

The above mentioned procedure counts on two “universal” mediation services. Obviously, there is nothing that prevents the creation and the usage of ooMediators, which count on specific mediation services, more or less tuned for specific ontologies. While such an approach is possible, one can hardly provide a generally applicable specification, methodology, and tool support for mediator development.

Run-time, the mediation process is determined by the WSMO execution environment, which decides whether and how the ooMediators are to be used. A possible example of the latter is given in section 5.2 of [14].

2.1 Component Specification

The Ontology Mapping Tool is not specified as long as it is not invoked directly – it is only used by a human to create a mapping. The Ontology Mapping Store is specified in section 4.2.

Pseudo-formal specifications of the rest of the components presented at Figure 4 are given below. Those consist of a Java-like method specification(s) and an information comment.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYTS Applicability Checker</td>
<td><strong>Boolean check</strong>(MappingID id)</td>
</tr>
<tr>
<td></td>
<td>Checks whether a correct syntactic transformation can be derived from a particular ontology mapping. This component could be integrated with the Semantic Transformation Generator or with the Rule Execution Engine (REE).</td>
</tr>
<tr>
<td>Semantic Transformation Generator</td>
<td><strong>SemTransfSpec generate</strong>(MappingID mapping)</td>
</tr>
<tr>
<td></td>
<td>A sort of a compiler of ontology mappings, which produces executable rules, as discussed in [4]. The specification format depends on the REE. This component can be integrated with REE; it could be unnecessary if REE can interpret mappings directly.</td>
</tr>
<tr>
<td>Syntactic Transformation Generator</td>
<td><strong>SyntTransfSpec generate</strong>(MappingID mapping)</td>
</tr>
<tr>
<td></td>
<td>A sort of a compiler of ontology mappings, which produces syntactic transformation rules. A specification of such a component that produces XSLT is provided in section 5.</td>
</tr>
<tr>
<td>Syntactic Transformation Service (SYTS)</td>
<td><strong>Instance mediate</strong>(SyntTransfSpec spec, Instance id)</td>
</tr>
</tbody>
</table>
|                                    | A web service, which should be specified accordingly; here
we only offer an idea of the input and the output. The implementation of the service can be based on a straightforward XSLT transformation.

<table>
<thead>
<tr>
<th>Semantic Transformation Service (SETS)</th>
<th>Instance mediate(SemTransfSpec spec, Instance id)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A web service, which should be specified accordingly; here we only offer an idea of the input and the output. The implementation of the service is expected to use the REE.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mediation Library</th>
<th>MediatorList lookup(Ontology source, Ontology target)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A mediator library, which allows for storage, retrieval, and lookup of mediators. In terms of Java API, the parameters should comply with the WSMO API, [12]. Such a component, including more advanced features as well, is scheduled for development within deliverable D5.6. Basic support for the lookup of mediators should also be provided through the Resource Manager (see section 4.8 of [14]).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule Execution Engine (REE)</th>
<th>Instance transform(SemTransfSpec spec, Instance id)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This engine can be expected to use a WSML reasoner. Many of the transformations can be regarded as, or transformed into, specific reasoning tasks. The formal language, which the mapping language is grounded to, might pose some restrictions on the use of several constructs of the mapping language, because such constructs are beyond the expressible power of the formal language.</td>
</tr>
</tbody>
</table>

|                             | Within SEKT, it is envisaged that REE uses an instance unification component – see the InstanceTransformation and InstanceUnification interfaces in Appendix B. For a more detailed explanation, refer to [4]. However, in the general data-mediation context considered in DIP, unification is not applicable, because it requires mediation that is bound to the recipient. |

2.2 Mediator Specification

The mediators in WSMO, WSMX, and DIP represent a description of specific mediation capabilities of a mediation service. We hereby provide a proposal for an extension of the ooMediator, as defined in the WSMO specification, [21]. New attributes are marked with a “New!” label in the Comment column. The attributes already defined in [21] are also included, together with their mapping to those in the latest version of WSMO (v.1.1, [22]).

<table>
<thead>
<tr>
<th>ooMediator</th>
<th>Attribute</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonFunctionalProperties</td>
<td>As for any WSMO element, those specify the title, author, etc. The name is changed in v.1.1 to hasNonFunctionalProperties.</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>importedOntologies</td>
<td>A specification of the imported ontologies, which does not require any conflict resolution.</td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>Entities, which are a source for the mediator, i.e. the input for mediation. In the case of an ooMediator for data-mediation - a source ontology. The name is changed in v.1.1 to hasSource.</td>
<td></td>
</tr>
<tr>
<td>target</td>
<td>Entities, which are a target for the mediator, i.e. the result of the mediation. In the case of an ooMediator for data-mediation - the target ontology. The name is changed in v.1.1 to hasTarget.</td>
<td></td>
</tr>
<tr>
<td>mediationService</td>
<td>Points to a goal or a web service, which actually implements the mapping. In the case of an ooMediator for data-mediation, it is a semantic or a syntactic transformation service (SETS or SYTS). The name is changed in v.1.1 to hasMediationService.</td>
<td></td>
</tr>
<tr>
<td>ontologyMapping</td>
<td>New! The ID of an ontology mapping, available from a mapping store.</td>
<td></td>
</tr>
<tr>
<td>transformationSpec</td>
<td>New! A transformation specification appropriate to the mediation service. In the case of a syntactic transformation, as defined in section 5, this is an XSLT. To make the storage, retrieval, and referencing of such specifications possible, either the Mapping Store or the Mediation Library should be extended with the appropriate interfaces and functionality.</td>
<td></td>
</tr>
</tbody>
</table>

There is not an immediate need for an extension of the WSMO specification to accommodate the above mentioned extensions. As a kick-off, those could be encoded as non-functional properties, as long as a concrete convention is adapted within the scope of the execution environment.

As a more general note, the mediators defined hereby could be considered a new-sort, data-to-data mediators, which could be abbreviated to ddMediators. It is not justified to extend the WSMO specification in this direction before the approach gets verified and matured in some pilot implementations.
3 Ontology Mapping Language

As mentioned above, the work in WP5 will rely on the ontology mapping language and the ontology mapping patterns developed in D4.3.1, [8], in SEKT. In this section we will give a short summary of this work. An important remark on the development of the language is made in sub-section 3.3.

The main design goal of the ontology language, developed within SEKT, was to capture general patterns in ontology mapping, independent of the ontology language used. However, due to the number of differences among existing ontology languages, this independence is not always possible. Therefore the developed language has a bias towards WSML-Flight [10] and OWL [11]. Also, as there are already significant differences between these two languages, only an abstract syntax for the mapping language is defined. The abstract syntax of the ontology mapping language can be found in Appendix B of [8]. In order to map two ontologies, e.g. two WSML ontologies, a grounding for the abstract mapping language has to be defined. However, the definition of this grounding, based on the abstract mapping language, is straightforward and is therefore not addressed in [8].

In its early phase, the WSMX project has used FLORA2 as a grounding language for the abstract mappings, [18]. A similar approach has also been adapted within the SWWS project, [16].

3.1 Ontology Mapping Patterns

An ontology is defined as a 4-tuple \(\{C, R, I, A\}\) with classes C, relations R, instances I, and axioms A. Based on this definition of an ontology, 25 ontology mapping patterns are identified. These patterns can be roughly divided into three groups: class mappings, relation mappings, and individual mappings. The attribute mappings are considered as a special case of relation mappings. Note that, while most of the patterns consider mappings between elements of one and the same kind (e.g. class-to-class), there are also cases when different elements are mapped. Such an example is the case when “student” is defined as a class in the source ontology, but as a Boolean attribute in the target one. An example of such mapping pattern is attributeClassMapping.

The list in Figure 5 shows the hierarchical organisation of the ontology mapping patterns. For a detailed description of each of the identified mapping patterns, consisting of a problem description, a description of the solution, and explanatory examples, see [8]. The main idea behind the identification and structuring of ontology mapping patterns is to guide the developer of ontology mappings into constructing correct ontology mappings. Further more, the identified patterns could also be used to guide the construction of ontology mapping algorithms. As the main focus of this deliverable is to specify a data mediation module, we are not going to elaborate further on the mapping patterns, but rather focus specifically on the mapping language derived from the patterns.

The identified ontology mapping patterns are used as a basis for the definition of the ontology mapping language. Note that the ontology mapping language does not contain a language primitive for each pattern but defines a minimal set of language primitives that can be combined in order to express each of the identified ontology mapping patterns.
1. **Class Mapping**
   a. Equivalent Class Mapping
   b. Subclass Mapping
   c. Class Intersection Mapping
      i. Equivalent Class Intersection Mapping
      ii. Subclass Intersection Mapping
   d. Class Union Mapping
      i. Equivalent Class Union Mapping
   e. Class by Attribute Mapping
   f. Class by Axiom Mapping
   g. Join Class Mapping
   h. Class Attribute Mapping
      i. Class Relation Mapping
      j. Class Individual Mapping

2. **Relation Mapping**
   a. Subrelation Mapping
   b. Equivalent Relation Mapping
   c. Attribute Mapping
      i. Attribute Transitive Closure Mapping
      ii. Attribute Inverse Mapping
      iii. Attribute Symmetry Mapping
      iv. Attribute Reflexivity Mapping
      v. Attribute Class Mapping
      vi. Attribute Value Mapping
         1. Equivalent Attribute Value Mapping
         2. Subattribute Value Mapping
   d. Relation Negation Mapping
      i. Subrelation Negation Mapping
   e. Relation by Axiom Mapping
   f. Join Relation Mapping
   g. Relation Class Mapping

3. **Individual Mapping**
   a. Equivalent Individual Mapping
   b. Equivalent Relation Instance Mapping
   c. Individual Class Mapping

**Figure 5. Hierarchical Organisation of Mapping Patterns**
3.2 Syntax of the Mapping Language

In this section we will briefly describe the syntax of the ontology mapping language. We will focus only on the parts of the syntax that are important for this specification, namely the keywords of the mapping language. A detailed description of the language, including abstract syntax in BMF format, can be found in [8].

The most important keywords of the mapping language are the mapping statements. There are four mapping statements available, namely `classMapping`, `relationMapping`, `attributeMapping`, and `individualMapping`. In addition to these mapping statements, the following keywords are available: `two-way`, `and`, `or`, `not`, `trans`, `inverse`, `symmetric`, and `reflexive`. The exact definitions of these keywords can be found in [8], too, therefore those are not discussed hereby in further details.

The following table shows how the statements in the mapping language are related to the mapping patterns, described in the previous section.

<table>
<thead>
<tr>
<th>Mapping Pattern</th>
<th>Mapping Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>equivalentClassMapping</td>
<td><code>classMapping(two-way C D)</code></td>
</tr>
<tr>
<td>subClassMapping</td>
<td><code>classMapping(C D)</code></td>
</tr>
<tr>
<td>equivalentClassIntersectionMapping</td>
<td><code>classMapping(two-way and(C1...Cn) D)</code></td>
</tr>
<tr>
<td>equivalentClassUnionMapping</td>
<td><code>classMapping(two-way or(C1...Cn) D)</code></td>
</tr>
<tr>
<td>subClassIntersectionMapping</td>
<td><code>classMapping(and(C1...Cn) D)</code></td>
</tr>
<tr>
<td>subClassUnionMapping</td>
<td><code>classMapping(or(C1...Cn) D)</code></td>
</tr>
<tr>
<td>subClassByAttributeMapping</td>
<td><code>classMapping(C D attributeOccurence(A))</code></td>
</tr>
<tr>
<td>subClassByAxiomMapping</td>
<td><code>classMapping(C D { axiom })</code></td>
</tr>
<tr>
<td>subRelationMapping</td>
<td><code>relationMapping(R D)</code></td>
</tr>
<tr>
<td>equivalentRelationMapping</td>
<td><code>relationMapping(two-way R D)</code></td>
</tr>
<tr>
<td>attributeTransitiveClosureMapping</td>
<td><code>attributeMapping(two-way A trans(B))</code></td>
</tr>
<tr>
<td>attributeInverseMapping</td>
<td><code>attributeMapping(two-way A inverse(B))</code></td>
</tr>
<tr>
<td>attributeSymmetricClosureMapping</td>
<td><code>attributeMapping(two-way A symmetric(B))</code></td>
</tr>
<tr>
<td>attributeReflexiveClosureMapping</td>
<td><code>attributeMapping(two-way A reflexive(B))</code></td>
</tr>
<tr>
<td>equivalentAttributeValueMapping</td>
<td><code>attributeValueMapping(two-way A I B J)</code></td>
</tr>
<tr>
<td>subAttributeValueMapping</td>
<td><code>attributeValueMapping(A I B J)</code></td>
</tr>
<tr>
<td>subRelationNegationMapping</td>
<td><code>relationMapping(R not(S))</code></td>
</tr>
<tr>
<td>subRelationByAxiomMapping</td>
<td><code>relationMapping(R S { axiom })</code></td>
</tr>
<tr>
<td>equivalentIndividualMapping</td>
<td><code>individualMapping(I J)</code></td>
</tr>
<tr>
<td>equivalentRelationInstanceMapping</td>
<td><code>individualMapping(R(II;... ; In)</code>)</td>
</tr>
</tbody>
</table>
Table 1: Correspondence between mapping patterns and the ontology mapping language

The mapping language statements, presented in the table above, will be the set of mapping statements supported by the data mediation component.

3.3 Evolution of the Mapping Language

The mapping language (as well as deliverable [8]) was still under development when this deliverable, and in particular this section, was written. By the time of final editing, there were already a number of changes, which make parts of this section incorrect or incomplete. While the mapping approach and many of the primitives remain the same, it is strongly recommended the final version of [8] to be consulted in case of formal reference, specification or implementation of tools related to it. This is particularly the case with the transformation specified in section 5.1.1.
4 Ontology Mapping-Related Interfaces

Here we present the ontology mediation component (OMC) specification, as it is defined in the SEKT project, [4]. OMC is specified in compliance with the ontology mapping language, defined in [8], which is also referred there as a mapping patterns library. There are four interfaces specified for this component:

- Pattern Store – maintenance of abstract patterns;
- Mapping Store – maintenance of concrete mappings (a definition of pattern and mapping are provided below);
- Instance Transformation;
- Instance Unification – used in the process of instance transformation.

The pattern and mapping store interfaces can be implemented on the basis of the specification of the abstract mapping language from [8]. The mapping store is the component which is directly relevant to the data-mediation environment, as specified in section 2 and Figure 4 there. The pattern store is necessary as long as the concrete mappings are defined as instantiations of patterns, thus the later are necessary for the definition of the mappings (e.g. within a mapping tool) and their interpretation.

The Instance Unification interface is only required for a specific approach for implementation of instance transformation. An implementation of the Instance Transformation interface could be the core of an implementation of a Semantic Transformation Service (see section 2). However, it have to be considered that an instance transformation implementation would require further grounding the mapping language into an “executable” logical formalism (e.g. F-Logic).

The mismatch of the components and the complex relationships between the mediation approach considered here and the OMC interfaces is caused on the fact that OMC is specified within SEKT for overlapping, but still different, purposes.

In this section we only focus on the data-model and the definition of the separate tasks, providing just a few of the most important functional calls. A more detailed definition of the APIs can be found in [4]. JavaDoc-like descriptions of the concrete interfaces can be found in Appendix B.

4.1 Ontology Mapping Data Model

As introduced in [8] and section 3.1 here, a pattern is considered a template expressed in the mapping language, not a concrete mapping between, say, two classes. Thus the patterns are abstract, i.e. not related to specific ontologies. A conceptual definition of the attributes of a pattern follows.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Attributes descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>The definition of the pattern specified in the terms of the pattern language, as given in [8].</td>
</tr>
<tr>
<td>Name</td>
<td>A short name or title, not necessarily unique.</td>
</tr>
<tr>
<td>Description</td>
<td>A human-readable informal description of the pattern.</td>
</tr>
</tbody>
</table>
Related Patterns | A list of references to patterns with respect to part-of, is-a or related-to semantic relations.
--- | ---
ID | An identifier, provided by the pattern store.

The concrete correspondences between elements in the source and target ontologies (e.g. two classes) are called *mappings*; those are instantiations of one or more patterns specified according to the mapping language. A conceptual definition of the attributes of a mapping follows.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Attributes descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>The definition of the mapping specified in the terms of the mapping language defined in [8]</td>
</tr>
<tr>
<td>Name</td>
<td>A short name or title, not necessarily unique</td>
</tr>
<tr>
<td>Description</td>
<td>A human-readable informal description of the pattern</td>
</tr>
<tr>
<td>Source Ontology; Target Ontology</td>
<td>References to the ontologies, which it is specified for; the ontology objects are defined through their URIs, while at the API level the WSMO API is used, [13]</td>
</tr>
<tr>
<td>Referenced Patterns</td>
<td>A list of references to template patterns, which have been used for the ontology mapping through instantiation</td>
</tr>
<tr>
<td>ID</td>
<td>An identifier, provided by the mapping store; still a matter of discussion</td>
</tr>
</tbody>
</table>

Example mapping:

```
Name: “o1_o2”
Definition: classMapping(o1:Person o2:Human
   attributeMapping(o1:name o2:name)
   attributeMapping(o1:age o2:age))
Version: “1.2.1”
Source and Target Ontologies: references to o1 and o2
Description: “Mapping between the ontologies o1 and o2”
Referenced patterns: references to the patterns:
   equivalenceClassMapping(classExpr  classExpr) and
equivalentRelationMapping(relationExpr  relationExpr)
```

4.2 Pattern and Mapping Store Interfaces

`PatternStore` and `MappingStore` interfaces are provided, with their obvious objectives. There could be multiple implementations for them, e.g. file-based, RDBMS-based or full-text. The `StorageFactory` class is used to create instances of these interfaces.

Patterns and Mappings can be loaded by ID. Those could also be searched on the basis of attribute restrictions, specified in detail in [4]. Removal of patterns and mappings is also defined. For consistency reasons, the store specification does not allow for the
deletion of patterns, which are in use (referenced) by mappings known to this store. Interface specification can be found in Appendix B.

4.3 Instance Transformation Interface

The *InstanceTransformation* interface allows for a fully automatic transformation of instances or sets of instances between two ontologies, according to a given mapping. The ontologies are specified in the mapping itself. Although semi-automatic and manual transformations are envisaged within SEKT, this interface covers the automatic modes only. There are two transformation modes defined:

- **batch mode** – a set of instances (or all instances) is transformed;
- **run-time mode** – only one instance (and possibly the related ones) is transformed.

4.4 Instance Unification Interface

An essential part of the process is the check whether the instance being transformed already exists in the target ontology. It is performed through the *InstanceUnification* interface, which “answers” whether two instances are identical (e.g. two different formal representations of one and the same city). For performance reasons, in the process of transforming instances, any unification on an instance-by-instance basis should be avoided. Therefore, the mediation component specification suggests approaches to prevent such an unification. One important assumption is that only instances of compatible classes can be identical. Compatibility, as well as a number of other details regarding the unification procedure, is discussed in some more details in [4].

Within the current specification of data mediation in DIP, it is considered that instance matching (the “local” term for an unification) is impossible, because the mediator may not have access to the data of the recipient. We hereby present this interface for reasons of exhaustiveness, only because it is part of the approach taken within SEKT. This mismatch is not critical, because a simple shortcut for adapting a SEKT mediation component in DIP is to define a dummy unification procedure that always fails.
5 CREATION OF SYNTACTIC TRANSFORMATION RULES

As stated in DIP Deliverable D5.1, the creation of direct transformations between a source and a target schema is an important requirement for the integration of Semantic Web Services technology into existing environments. In principle, these direct transformations could be created either during design-time or during run-time. Each of the two approaches has different implications and faces different open research issues, which will be outlined in the following sections. The basic scenario for data-mediation within the overall mediation approach, specified in section 2, only considers the design-time creation of transformations. Still, a scenario for automatic run-time generation of syntactic transformation from an existing ontology mapping is feasible, e.g. as a failover strategy for cases when a semantic transformation is impossible.

In order to express the syntactical transformation rules, we will use XSLT.

5.1 Creation of syntactic transformations during design-time

The creation of direct transformations during design-time would require some software component, which takes a number of ontology transformations, a lifting, and a lowering, and which compiles a single transformation from it. In case both or one of the parties is already WSML-speaking (natively, or because it is configured with an adapter), the corresponding lifting/lowering phase is found to be obsolete. This is the main scenario, recommended in the general data mediation approach proposed hereby.

It is important to note that this approach offers the easiest integration into existing environments, since only the design-time part of the DIP architecture would be necessary. After a transformation is created in the design-time tool, it could simply be deployed into the existing infrastructure without the need to modify this infrastructure.

In the following paragraphs we are going to focus our discussion on the creation of XSLT scripts, however this is not a limitation to the approach because all presented ideas can be easily customized to support other syntactic transformation languages.

The simplest solution to this problem is to express each step of the transformation as a single XSLT script and to use a pipelining approach to generate the complete transformation (Figure 6). For each of the steps (Lifting, Mediation, and Lowering) a separate XSLT script is generated. During runtime, the input-data is transformed by executing the scripts one after another.
To enable the approach outlined above, it is necessary to determine how the primitive building-blocks can be transformed into XSLT constructs. Once this is specified, the XSLT transformation scripts can easily be created automatically for the lifting, mediation, and lowering descriptions.

An important note here is that the XSLT script will only be able to transform an instance of one ontology into an instance of the second ontology.

### 5.1.1 Language-primitive translations

In this section we will define how each of the language-primitives of the ontology mapping language can be mapped to an XSLT construct. As mentioned above, we are going to use WSML as an ontology language and the WSML XML syntax as an exchange format for ontologies. Therefore we will present how the mappings language primitives can be translated into XSLT constructs that perform a transformation between two WSML ontologies. However it will not be difficult to create similar XSLT constructs for another ontology language, e.g. OWL, based on the constructs presented, in case this proves necessary.

In addition to the translations of the language primitives, presented in this section, Appendix A shows an example of a mapping script and the corresponding translation.

It is important to note that the following tables only present the interesting parts of each mapping. The parts of the translation that are straightforward are not presented hereby. Examples of such parts of the translation are mappings of the nonfunctional requirements that are identical for each of the translations or mappings that simply copy content.

#### Translation of One-to-one Inter-class Mapping

| Mapping Statement | classMapping( two-way Class1 Class2 ) |

---

Figures 6. Use of pipelining to create syntactic transformations
### XSLT Translation

```xml
<xsl:template name="Class1toClass2Mapping"
    match="wsml:instance[wsml:memberOf = 'Class1']">
    <xsl:element name="wsml:instance">
        <xsl:attribute name="name">
            <xsl:value-of select="./@name"/>
        </xsl:attribute>
        <xsl:element name="wsml:memberOf">
            Class2
        </xsl:element>
        ...  
        Mapping of nonfunctional properties 
        ... 
        Attribute mappings 
        ...
    </xsl:element>
</xsl:template>
```

### Translation of Subclass/Superclass Mapping

**Syntax**

```plaintext```
classMapping( Class1 Class2 )
```

**XSLT Translation**

Translation is the same as the one for the One-to-one Inter-class Mapping.

### Translation of Equivalent Class Intersection Mapping

**Syntax**

```plaintext```
classMapping( two-way and(Class1,...,ClassN) ClassX )
```

**XSLT Translation**

```xml
<xsl:template name="Class1and...andClassNtoClassXMapping"
    match="wsml:instance[wsml:memberOf = 'Class1' and ... and wsml:memberOf = 'ClassN']">
    <xsl:element name="wsml:instance">
        <xsl:attribute name="name">
            <xsl:value-of select="./@name"/>
        </xsl:attribute>
        <xsl:element name="wsml:memberOf">
            Class2
        </xsl:element>
        ...  
        Mapping of nonfunctional properties 
        ... 
        Attribute mappings 
        ...
    </xsl:element>
</xsl:template>
```

### Translation of Equivalent Class Union Mapping

**Syntax**

```plaintext```
classMapping( two-way or((Class1,...,ClassN) ClassX) )
```

**XSLT Translation**

```xml
<xsl:template name="Class1or...orClassNtoClassXMapping"
    match="wsml:instance[wsml:memberOf = 'Class1']">
    <xsl:element name="wsml:instance">
        <xsl:attribute name="name">
            <xsl:value-of select="./@name"/>
        </xsl:attribute>
        <xsl:element name="wsml:memberOf">
            Class2
        </xsl:element>
        ...  
        Mapping of nonfunctional properties 
        ... 
        Attribute mappings 
        ...
    </xsl:element>
</xsl:template>
```
ClassX
</xsl:element>
...
Mapping of nonfunctional properties
...
Attribute mappings
...
</xsl:element>
</xsl:template>
...
<xsl:template
name="equivalentClassUnionMappingClassNtoClassX"
match="wsml:instance[wsml:memberOf  = 'ClassN']"
>
<xsl:element name="wsml:instance" >
<xsl:attribute name="name">
<xsl:value-of select="./@name"/>
</xsl:attribute>
<xsl:element name="wsml:memberOf">ClassX</xsl:element>
</xsl:element>
...
Mapping of nonfunctional properties
...
Attribute mappings
...
</xsl:element>
</xsl:template>

Translation of Subclass Intersection Mapping

Syntax
classMappinig( and(Class1,…,ClassN) ClassX)

XSLT Translation
Translation is the same as the one for the Equivalent Class Intersection Mapping.

Translation of Subclass Union Mapping

Syntax
classMappinig( or(Class1,…,ClassN) ClassX)

XSLT Translation
Translation is the same as the one for the Equivalent Class Union Mapping.

Translation of Equivalent Relation Mapping

Syntax
relationMappinig( two-way Relation1 Relation2)

XSLT Translation
<xml:template
name="equiRelationMappingRelation1toRelation2"
match="wsml:relationInstance[wsml:memberOf  = 'Relation1']">
<xsl:element name="wsml:relationInstance" >
<xsl:attribute name="name">
<xsl:value-of select="./@name"/>
</xsl:attribute>
<xsl:element name="wsml:memberOf">Relation2</xsl:element>
</xsl:element>
...
parameter value mappings
...
### Translation of Subrelation Mapping

**Syntax**

```
relationMapping( R S)
```

**XSLT Translation**

Translation is the same as the one for the Equivalent Relation Mapping.

### Subclass by Attribute Mapping

**Syntax**

```
classMapping( Class1 Class2 attribValueCond)
```

**XSLT Translation**

```
<xsl:template name="Class1toClass2MappingByattribCond"
            match="wsml:instance[wsml:memberOf = 'Class1']">
  <xsl:if test="/wsml:attributeValue/wsml:name[. = 'attrib'] and ./wsml:attributeValue/wsml:value[. = 'value']">
    <xsl:element name="wsml:instance">
      <xsl:attribute name="name">
        <xsl:value-of select="./@name"/>
      </xsl:attribute>
      <xsl:element name="wsml:memberOf">
        Class2
      </xsl:element>
    </xsl:element>
  </xsl:if>
</xsl:template>
```

### Subclass by Axiom Mapping

**Syntax**

```
classMapping( Class1 Class2 {axiom} )
```

**XSLT Translation**

This expression *cannot* be translated to XSLT as it is not possible to evaluate logical axioms without the usage of a reasoner. Therefore mappings containing this expression cannot be translated to XSLT.

In contrast to the mappings expressions, presented above, the mapping expression involving attributes cannot occur as a standalone one but is always a part of a class mapping expression. Therefore the translations of this expression are not complete XSLT templates, but only fragments that can be reused in the class mapping templates whenever necessary.

### Transitive Closure Mappings

**Syntax**

```
All expressions containing the trans keyword
```
XSLT Translation

Transitive closure cannot be computed using XSLT. Therefore it is not possible to translate any expression containing the trans keyword. As a result, mappings containing the trans keyword cannot be translated to XSLT.

<table>
<thead>
<tr>
<th>Equivalent Attribute Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
</tbody>
</table>
| **XSLT Translation** | `<xsl:element name="wsml:attributeValue" >
  `<xsl:element name="wsml:name" >
    Attrib2
  </xsl:element>
  `<xsl:element name="wsml:value" >
    `<xsl:value-of select="./wsml:attributeValue[wsml:value = "Attrib1" ]/wsml:value />
  </xsl:element>
</xsl:element>` |

<table>
<thead>
<tr>
<th>Inverse Attribute Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>XSLT Translation</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equivalent Attribute Value Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
</tbody>
</table>
| **XSLT Translation** | `<xsl:if test="./wsml:attributeValue/wsml:name[. = 'Attrib1'] and ./wsml:attributeValue/wsml:value[. = 'Value1']">`
| | `<xsl:element name="wsml:attributeValue" >
  `<xsl:element name="wsml:name" >
    Attrib2
  </xsl:element>
  `<xsl:element name="wsml:value" >
    Value2
  </xsl:element>
</xsl:element>` |

<table>
<thead>
<tr>
<th>Subattribute Value Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>XSLT Translation</strong></td>
</tr>
</tbody>
</table>
5.2 Creation of syntactic transformations during run-time

In contrast to creating the direct transformations already during design-time, the creation of them during run-time can be seen as an optimization technique. The mediation component would track which ones of the transformations are frequently used, and would optimize these transformations dynamically by creating a direct transformation.
6 CONCLUSION

The main objective of this deliverable has been to provide a specification for a data-mediation component. It first provides a structure and context to the data-mediation problem by making the distinctions, which have already been initiated in [6], more specific:

- design- vs. run-time mediation;
- syntactic vs. semantic transformation;
- relations to format adapters;
- relations to and usage of ontology mediation approaches, specifications, and technologies.

The main constraints and dependencies are hereby outlined and established onto the evolution path through the simple principle that the design-time mediation should not be fully automatic, and that there should be a human involved in the design loop in order to assure the maximum (possible) correctness of the mappings – as a result, a fully-automatic run-time mediation is allowed for.

Further, a number of related tools and activities are hereby analysed and put together:

- the ontology mapping language;
- the ontology mapping and pattern store;
- the corresponding WSMX elements.

An overall approach for data-mediation is outlined in section 2, together with the roles and high-level specifications for the related tools and components, including: an ontology mapping tool, a semantic transformation engine, mediator library.

A number of related technologies and concepts are still missing or immature – most notably, the ontology mapping language has been developed in parallel, and the ontology mapping tools, pattern stores, and execution engines are not available. In effect, this means that this deliverable provides guidance to the development of a data mediation toolset, but however not a complete specification that would allow for a straightforward implementation.
REFERENCES


APPENDIX A – EXAMPLE OF TRANSLATION OF ONTOLOGY MAPPING TO XSLT

This appendix will show an example translation from the ontology mapping language to XSLT. This example is based on the example presented in [7].

Example Mapping

```
classMapping (o1:Person o2:Human
    attributeMapping (o1:name o2:name)
    attributeMapping (o1:age o2:age)
  )

classMapping (o1:Person o2:Man
    attributeValueCondition(hasGender "Male")
  )
```

XSLT Translation

```
<xsl:template name="PersonToHumanMapping"
  match="wsml:instance[wsml:memberOf = 'Person']">
    <xsl:element name="wsml:instance">
      <xsl:attribute name="name">
        <xsl:value-of select="./@name"/>
      </xsl:attribute>

      <xsl:element name="wsml:memberOf">
        Human
      </xsl:element>

      <xsl:element name="wsml:attributeValue">
        <xsl:element name="wsml:name">
          name
        </xsl:element>
        <xsl:element name="wsml:value">
          <xsl:value-of select="./wsml:attributeValue[. = 'name']/wsml:value"/>
        </xsl:element>
      </xsl:element>

      <xsl:element name="wsml:attributeValue">
        <xsl:element name="wsml:name">
          age
        </xsl:element>
        <xsl:element name="wsml:value">
          <xsl:value-of select="./wsml:attributeValue[. = 'age']/wsml:value"/>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>

<xsl:template name="PersonToManMappingByAttributeCond"
  match="wsml:instance[wsml:memberOf = 'Person']">
    <xsl:if test="./wsml:attributeValue/wsml:name[. = 'gender'] and
      ./wsml:attributeValue/wsml:value[. = 'Male']">
```

```
<xsl:element name="wsml:instance" >
  <xsl:attribute name="name">
    <xsl:value-of select="./@name"/>
  </xsl:attribute>
  <xsl:element name="wsml:memberOf">
    Man
  </xsl:element>
</xsl:element>
</xsl:if>
</xsl:template>
APPENDIX B – ONTOLOGY MEDIATION COMPONENT APIs

This appendix provides descriptions of the StoreFactory, PatternStore, MappingStore, InstanceTransformation, and InstanceUnification interfaces in a JavaDoc-like notation. The descriptions are taken from [4]; there one can also find the ones for the remaining interfaces (Pattern, Mapping, the restriction-related ones).

StoreFactory

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>static MappingStore getMappingStore()</td>
</tr>
<tr>
<td>static PatternStore getPatternStore()</td>
</tr>
</tbody>
</table>

PatternStore

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern loadPattern(PatternID pattern)</td>
</tr>
<tr>
<td>Loads a pattern through its identifier</td>
</tr>
<tr>
<td>void removePattern(PatternID pattern)</td>
</tr>
<tr>
<td>Removes a pattern specified by its identifier.</td>
</tr>
<tr>
<td>Iterator searchByIsARelation(Pattern pattern)</td>
</tr>
<tr>
<td>Searches for all patterns, which have relation of type is-a with a given pattern</td>
</tr>
<tr>
<td>Iterator searchByKeywords(List keywords)</td>
</tr>
<tr>
<td>Searches for all patterns with given keywords in their descriptions</td>
</tr>
<tr>
<td>Iterator searchByName(String patternName)</td>
</tr>
<tr>
<td>Searches for a pattern with a given name</td>
</tr>
<tr>
<td>Iterator searchByPartOfRelation(Pattern pattern)</td>
</tr>
<tr>
<td>Searches for all patterns, which have a relation of type part-of with a given pattern</td>
</tr>
<tr>
<td>Iterator searchByRelatedToRelation(Pattern pattern)</td>
</tr>
<tr>
<td>Searches for all patterns, which have a relation of type related-to with a given pattern</td>
</tr>
</tbody>
</table>
### MappingStore

#### Method Summary

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loadMapping(MappingID id)</code></td>
<td>Loads a mapping through its identifier</td>
</tr>
<tr>
<td><code>removeMapping(MappingID mapping)</code></td>
<td>Removes a mapping specified by its identifier.</td>
</tr>
<tr>
<td><code>searchByKeywords(List keywords)</code></td>
<td>Searches for all mappings with the given keywords in their descriptions</td>
</tr>
<tr>
<td><code>searchByName(String name)</code></td>
<td>Searches for an ontology mapping with a given name</td>
</tr>
<tr>
<td><code>searchByPatterns(List patterns)</code></td>
<td>Searches for ontology mappings in respect to their list of related patterns</td>
</tr>
<tr>
<td><code>searchByRestriction(MappingRestriction mappingRestriction)</code></td>
<td>Searches for ontology mappings in respect to a given composite restriction</td>
</tr>
<tr>
<td><code>searchBySourceOntology(Identifier sourceOnt)</code></td>
<td>Searches for ontology mappings with specified source ontology</td>
</tr>
<tr>
<td><code>searchByTargetOntology(Identifier targetOnt)</code></td>
<td>Searches for ontology mappings with a specified target ontology</td>
</tr>
<tr>
<td><code>searchByVersion(Version v)</code></td>
<td>Searches for an ontology mapping with a given version</td>
</tr>
<tr>
<td><code>storeMapping(Mapping mapping)</code></td>
<td>Stores a pattern and returns a newly created pattern identifier</td>
</tr>
</tbody>
</table>

**Iterator**

- `searchByRestriction(PatternRestriction patternRestriction)`
  - Searches for patterns in respect to a given composite restriction
- `storePattern(Pattern pattern)`
  - Stores a pattern and returns a newly created pattern identifier
Stores a mapping and returns a newly created mapping identifier

### InstanceTransformation

**Method Summary**

<table>
<thead>
<tr>
<th>Collection</th>
<th>transformation(Mapping mapping, Collection instances, InstanceUnification iu)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Transforms a set of instances of one ontology to the corresponding instances of a second ontology according to a given mapping. The <strong>InstanceUnification</strong> parameter is the particular instance unification that has to be used when transforming instances.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance</th>
<th>transformation(Mapping mapping, Instance i, InstanceUnification iu)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Transforms an instance of one ontology to an instance of a second ontology according to a given mapping. The instance unification parameter is the same as in the above mentioned method.</td>
</tr>
</tbody>
</table>

### InstanceUnification

**Method Summary**

<table>
<thead>
<tr>
<th>Ontology</th>
<th>getTargetOntology()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns a reference to the target ontology, for which this instance unification is applicable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int</th>
<th>isTheSame(Instance i1, Instance i2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Checks whether two instances of the target ontology are the same. Returns the value 0 if it is not possible to give an answer, the value 1 if they refer to the same real object, or -1 in any other case.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance</th>
<th>merge(Instance i1, Instance i2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unifies two instances of the target ontology, creating a new instance in the place of the first one. It is possible to make a unification only if <strong>isTheSame()</strong> method returns 1 and so a reference to the newly created instance (first one) will be returned.</td>
</tr>
</tbody>
</table>