EU-IST Project IST-2003-506826 SEKT

SEKT: Semantically Enabled Knowledge Technologies

D4.5.1 Report on Ontology mediation as service component

Dirk Wenke (ontoprise)
Stoyan Atanassov (Sirma AI) Dimitar Manov (Sirma AI)
Mika Maier-Collin, Wolfgang Sperling (ontoprise)

Abstract

This document describes the mapping and its functionalities. An outlook of possible szenarios for the integration of mapping store, mapping components and mapping patterns is made.
A concept of an API is introduced and the expected usage is described.

Keyword list: ontology mediation, visualisation, graphical interface, mapping, pattern, store, instance transformation, instance unification

WP4 Ontology Mediation	
Report	PU
Contractual date of delivery	31.12.2004
Actual date of delivery	31.01.2004

SEKT Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2003-506826.

British Telecommunications plc.
Orion 5/12, Adastral Park
Ipswich IP5 3RE
UK
Tel: +44 1473 609583, Fax: +44 1473 609832
Contact person: John Davies
E-mail: john.nj.davies@bt.com

Empolis GmbH
Europaallee 10
67657 Kaiserslautern
Germany
Tel: +49 631 303 5540
Fax: +49 631 303 5507
Contact person: Ralph Traphöner
E-mail: ralph.traphoener@empolis.com

Jozef Stefan Institute
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 4773 778, Fax: +386 1 4251 038
Contact person: Marko Grobelnik
E-mail: marko.grobelnik@ijs.si

University of Karlsruhe, Institute AIFB
Englerstr. 28
D-76128 Karlsruhe
Germany
Tel: +49 721 608 6592
Fax: +49 721 608 6580
Contact person: York Sure
E-mail: sure@aifb.uni-karlsruhe.de

University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1891
Fax: +44 114 222 1810
Contact person: Hamish Cunningham
E-mail: hamish@dcs.shef.ac.uk

University of Innsbruck
Institute of Computer Science
Technikerstraße 13
6020 Innsbruck
Austria
Tel: +43 512 507 6475
Fax: +43 512 507 9872
Contact person: Jos de Bruijn
E-mail: jos.de-bruijn@deri.ie

Intelligent Software Components S.A.
Pedro de Valdivia, 10
28006
Madrid
Spain
Tel: +34 913 349 797
Fax: +49 34 913 349 799
Contact person: Richard Benjamins
E-mail: rbenjamins@isoco.com

Kea-pro GmbH
Tal
6464 Springen
Switzerland
Tel: +41 41 879 00
Fax: 41 41 879 00 13
Contact person: Tom Bösser
E-mail: tb@keapro.net

Ontoprise GmbH
Amalienbadstr. 36
76227 Karlsruhe
Germany
Tel: +49 721 50980912
Fax: +49 721 50980911
Contact person: Hans-Peter Schnurr
E-mail: schnurr@ontoprise.de

Sirma AI EAD, Ontotext Lab
135 Tsarigradsko Shose
Sofia 1784
Bulgaria
Tel: +359 2 9768 303, Fax: +359 2 9768 311
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

Vrije Universiteit Amsterdam (VUA)
Department of Computer Sciences
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
Tel: +31 20 444 7731, Fax: +31 84 221 4294
Contact person: Frank van Harmelen
E-mail: frank.van.harmelen@cs.vu.nl

Universitat Autonoma de Barcelona
Edifici B, Campus de la UAB
08193 Bellaterra (Cerdanyola del Vall` es)
Barcelona
Spain
Tel: +34 93 581 22 35, Fax: +34 93 581 29 88
Contact person: Pompeu Casanovas Romeu
E-mail: pompeu.casanovas@uab.es

Executive Summary

State of the Art and challenges

The knowledge management issue becomes more and more complex due to the existence of heterogeneous information and communication channels and the dynamic feature of the life. There does not exist THE Ontology which could improve the current situation. Therefore ontology mediation is a very important part for the whole ontology management structure. Current solutions for ontology mediation still stay at the stage of manually aligning and mapping ontologies with some limited recommendation services. In order to scale up the whole process, semi-automatic and automatic methods for ontology mediation are a big challenge. In particular, providing ontology mediation as a plug-in-and-play software component is our goal.

Justification

Knowledge management pursues the efficient process of creating, presenting, communicating and reusing knowledge with complex organizations. While these complex organizations are typically structured in different components and operated with a high degree of autonomy. Managing knowledge within autonomous groups and exchanging knowledge across them are the big challenge at the moment.

Due to the existing heterogeneous nature and different requirements derived from the applications and tasks, there will co-exist various different views on the same information or knowledge. Mediation is clearly well demanded for communicating and reusing knowledge. Mediation makes communicating and reusing knowledge possible without losing its semantics or altering it. It is akin to two people speaking different languages needing to talk, and the dictionary containing both languages being the essential enabler.

Knowledge management is focusing on knowledge use and reuse. Mediation is a kind of knowledge, therefore how to manage it and further reuse it is important to provide the efficient and effective mediation services.

Ontology Mediation as Service Component

Ontology mediation can be provided as a service component to easily plug-in the existing knowledge management platform. For instance, mediation can be defined by the users, therefore an ontology mediation authoring environment provides the access point for the users to define their own mediation.

For the first step a visualisation of the mapping is needed for allowing the user to interactively map an ontology to another ontology. This enables the user to pose a query to the original ontlogy and get the appropriate answers which consider the input from other ontologies. These ontologies could be integrated by manually modelling ontologies or by importing of schemas from relational databases.

A Mapping and Pattern data model is defined, then the Mapping and Pattern stores are described. Finally an approach to solving instance transformation and instance unification problems is presented.

The integration in the SEKT architecture is enabled by the strictly usage of patterns, defined in the deliverable D4.4.1[1] .

Overall Picture

The overall picture is shown in fig. 1. The ontologies as well as the mappings between ontologies are stored in ontology repositories and mapping repositories. These repositories may be accessed by different tools. E.g. mapping discovery tools like those developed in WP 4.4.1 feed their results into that repositories. OntoStudio accesses these repositories to retrieve ontologies, corresponding mappings and mapping patterns. OntoStudio visualizes these mappings and makes them creatable and changeable in an interactive graphical way by the user. All these components are communciation via webservices, thus building a service-based architecture.

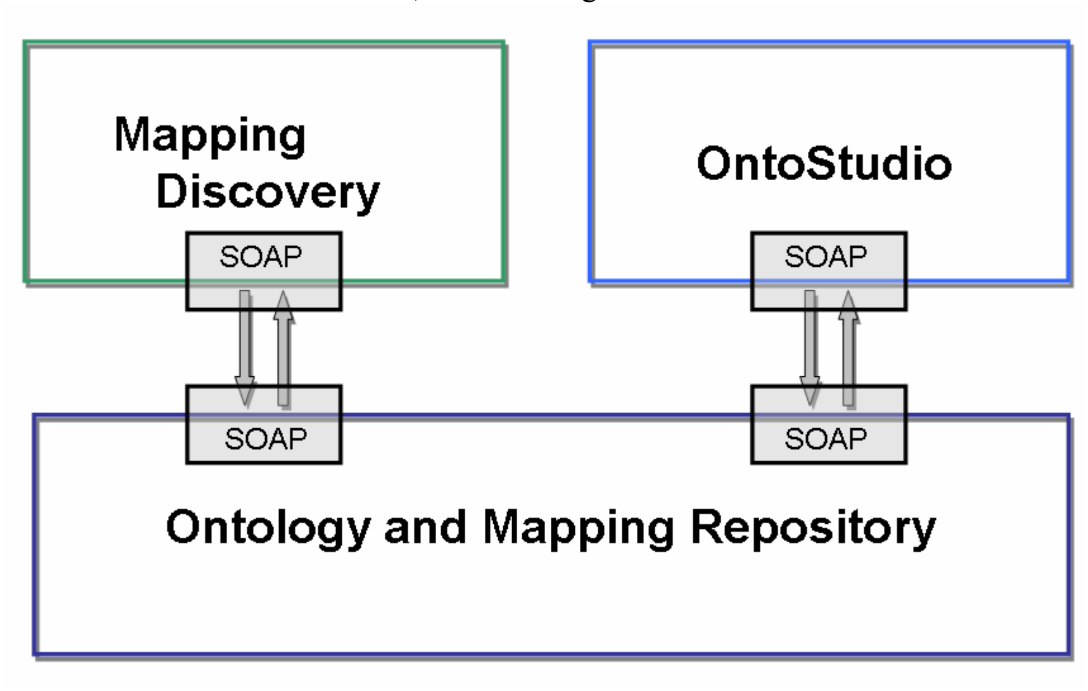

Figure 1: Overall Picture

Chapter 1 describes the technical foundation of mappings in OntoStudio. In chapter 2 the graphical user interface in OntoStudio is shown. Chapter 3 briefly presents the glue between the the different components already mentioned in fig. 2. Finally in chapter 4 the ontology and mapping repository together with its API is described.

Contents

SEKT Consortium **2**
Executive Summary **3**
State of the Art and challenges 3
Justification 3
Ontology Mediation as Service Component 3
Overall Picture 4
1 Technical description of mapping **6**
1.1 OntoMap – Mapping Plugin for OntoStudio 6
2 Graphical interface of mapping **9**
2.1 Concepts on concepts 10
2.2 Attributes on concepts 11
2.3 Attributes on attributes 12
2.4 Relations on relations 13
3 Integration of the different components **13**
3.1 Mapping store – Mapping Component 13
3.2 Mapping patterns – Mapping Component 14
4 Component Description **14**
4.1 Data Model 14
4.1.1 Pattern 14
4.1.2 Mapping 14
4.2 Store 15
4.2.1 Pattern and Mapping Store 15
4.2.2 Pattern and Mapping Restrictions 16
4.3 Instance Transformation 17
4.4 Instance Unification 17
4.4.1 Cascading unifications 18
5 API **20**
5.1 Data Model 20
5.1.1 Pattern 20
5.1.2 Mapping 21
5.1.3 Version 21
5.2 Store 23
5.2.1 Pattern Store 23
5.2.2 Mapping Store 23
5.2.3 Storage Factory 24
5.2.4 Pattern/Mapping Restriction 24
5.2.5 Example Scenarios 27
5.3 Instance Transformation 31
5.4 Instance Unification 31
6 Bibliography and references **32**

1 Technical description of mapping

There are two different scenarios for the usage of a mapping functionality.
The first one is to retrieve instances for an ontology from a given database scheme, the second one is to map two different ontologies to define where they have common concepts and relations.
OntoStudio contains an instance editor that can be used by the ontology engineer to create test sets. Another way to get test instances is to fetch them from a database. Therefore OntoStudio is able to import the relational schema of a database and create a (flat) ontology out of that schema. Mapping allows the user to interactively map an ontology to another ontology. The relationships between the mapped ontologies are formally represented by F-Logic axioms. In this way the original ontology may be populated with instances out of the database. If now a query is posed to the original ontology SQL queries are generated at runtime to get the appropriate answers for the query out of the database.
Mapping two ontologies has a similar meaning and a similar effect. If a mapping between two concepts is defined and a query is formulated against the destination ontology (i.e. the ontology the other ontology is mapped to) the rules that have been formulated transform the query to a query against the source ontology (i.e. the original ontology). This will allow the reuse of parts of ontologies (with their possible database mapping in the background) as well as the definition of different conceptual views.

1.1 OntoMap – Mapping Plugin for OntoStudio

OntoMap is a plugin of OntoStudio which allows to manually map ontologies. It allows to graphically draw arrows between several source and one target ontology and thus to define these mappings. Mappings may be created from:

1. concept C_1 to concept C_2
2. attribute A_1 to attribute A_2
3. relation R_1 to relation R_2
4. attribute A to concept C

OntoMap interprets these mappings by generating the following F-Logic mapping rules:

1. concept C_1 to concept C_2: FORALL X X:C_2 <- X:C_1.
2. attribute A_1 to attribute A_2: FORALL X,Y X:C_2 [A_2->Y] <- X:C_1[A_1->Y].
3. relation R_1 to relation R_2: FORALL X,Y X:C_2 [R_2->Y] <- X:C_1[R_1->Y].
4. attribute A to concept C: FORALL X,Y Y:C <- X:C_1[A->Y].

It is obvious that for mappings 2 and 3 the concepts C_1 and C_2 have to be mapped before. The fourth mapping means that the attribute values of attribute A are the ids for the instances of class C.

For the further processing of the mappings and its rules each mapping is represented as a mapping-instance. A mapping consists of a target and several sources, a relation to the generated rule and optional to mapping-conditions and -functions:

- Mapping[

  Name=>STRING;
  Description=>STRING;
  SourceOntology=>URI;
  TargetOntology=>URI;
  hasRule=>Rule;
  hasCondition=>MappingCondition;
  hasFunction=>Function].

- MappingCondition[

  CompareAttribute=>Attribute;
  CompareOperator=>Operator;
  CompareValue=>STRING].

- ConceptToConceptMapping ::Mapping.

  ConceptToConceptMapping[
  SourceConcept=>Concept;
  TargetConcept=>Concept].

- AttributeToConceptMapping::Mapping.

  AttributeToConceptMapping [
  SourceAttribute=>Attribute;
  SourceConcept=>Concept;
  TargetConcept=>Concept].

- AttributeToAttributeMapping::Mapping.

  AttributeToAttributeMapping [
  SourceAttribute=>Attribute;
  SourceConcept=>Concept;
  TargetAttribute=>Attributet;
  TargetConcept=>Concept].

- RelationToRelationMapping::Mapping.

  RelationToRelationMapping [
  SourceRelation=>Relation;
  SourceConcept=>Concept;
  TargetRelation=>Relation;
  TargetConcept=>Concept].

These patterns can be compared with patterns described in deliverable D.4.3.1 except mapping-instances created with OntoMap are always unidirectional. When bidirectional mappings are needed, two mappings have to be defined (one from A to B and vice versa). Furthermore not all possible patterns are realized within OntoMap, in fact only patterns supported by the graphical interface actually are used.

Nevertheless most patterns described in deliverable D.4.3.1 are covered by OntoMap.

"ConceptToConceptMapping" covers:

- 5.1.2 "Subclass/Superclass Mapping" as direct equivalent,

- 5.1.1 “Equivalent Classes” and
- 5.1.4 “Class Union” except for bidirectional mappings,
- 5.1.5 “Class by Attribute Mapping” and
- 5.4 “Attribute Value-Class Equivalence” by applying mapping conditions.

“AttributeToConceptMapping” covers:
- 5.1.8 “Class Attribute Mapping” except mapping from concepts to attributes

“AttributeToAttributeMapping” and “RelationToRelationMapping” cover
- 5.2.1 “Equivalent Relation Mapping” and
- 5.2.2 “Subrelation-Superrelation Mapping.
- 5.2.3 “Negated Relation Mapping”,
- 5.2.5 “Attribute Transitive Closure” and
- 5.2.6 “Inverse Attribute Mapping” can be supported by expressing functions.

In further versions of OntoMap the following patterns could be supported by graphical means:

5.1.3 “Class Intersection”
5.1.7 “Class Join Mapping”
5.2.7 “Attribute Value Mapping”
5.3.1 “Equivalent Individual Mapping”
5.3.2 “Equivalent Relation Instance Mapping”

The following patterns are too complex to be supported by a graphical tool and therefore could only be supported by a rule editor:

5.1.6 “Class Mapping by Axiom”,
5.1.9 “Class Relation Mapping”,
5.1.10 “Class Instance Mapping”,
5.2.4 “Relation Mapping by Axiom”.

2 Graphical interface of mapping

OntoMap extends OntoStudio with the ability to map data structures between ontologies. It comes along with a mapping view (see Figure 2) in which mappings can be defined between several source ontologies (on the left) and one target ontology (on the right). A mapping is visualised by an arrow and can be created by a drag&drop-operation from the source to the target.

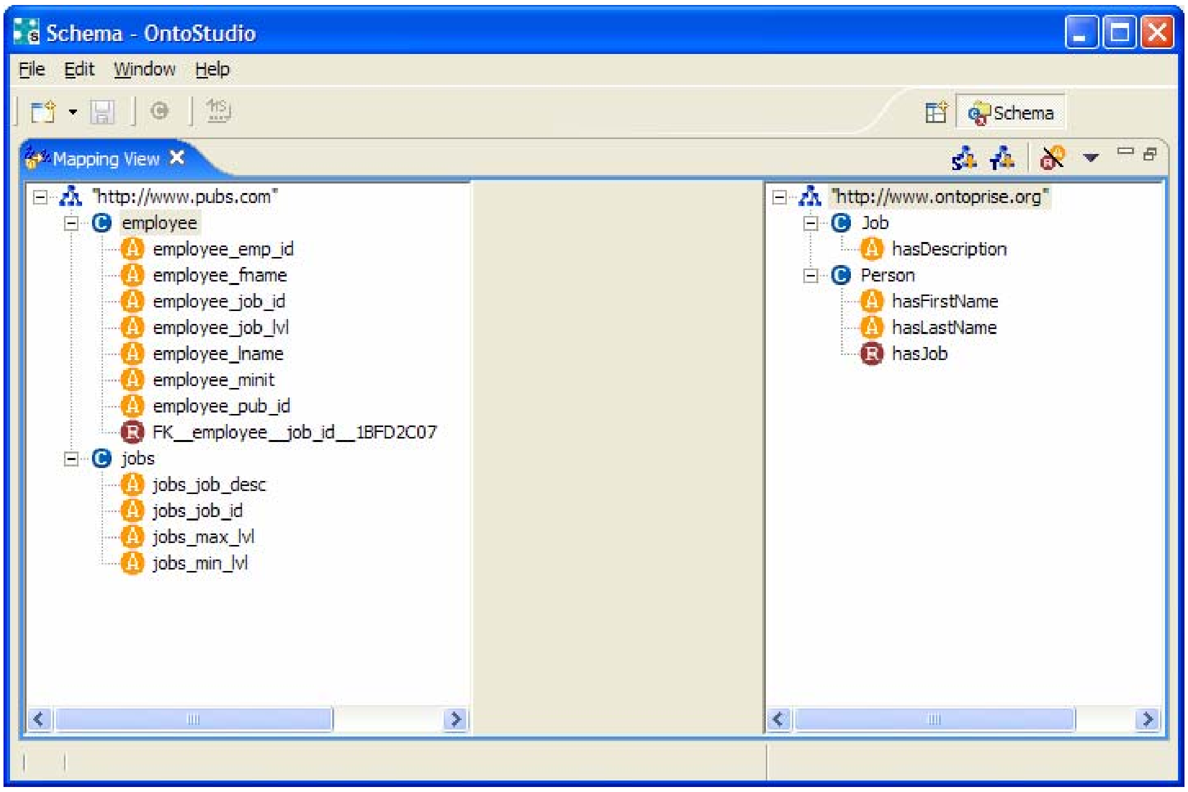

Figure 2: OntoStudio with Mapping View

OntoMap offers several mapping possibilities:

- concepts on concepts
- attributes on concepts
- attributes on attributes
- relations on relations

These mapping possibilities are described next by examples using the mapping scenario shown in 2, with a flat ontology from a database on the left and a target/enterprise-ontology on the right.

2.1 Concepts on concepts

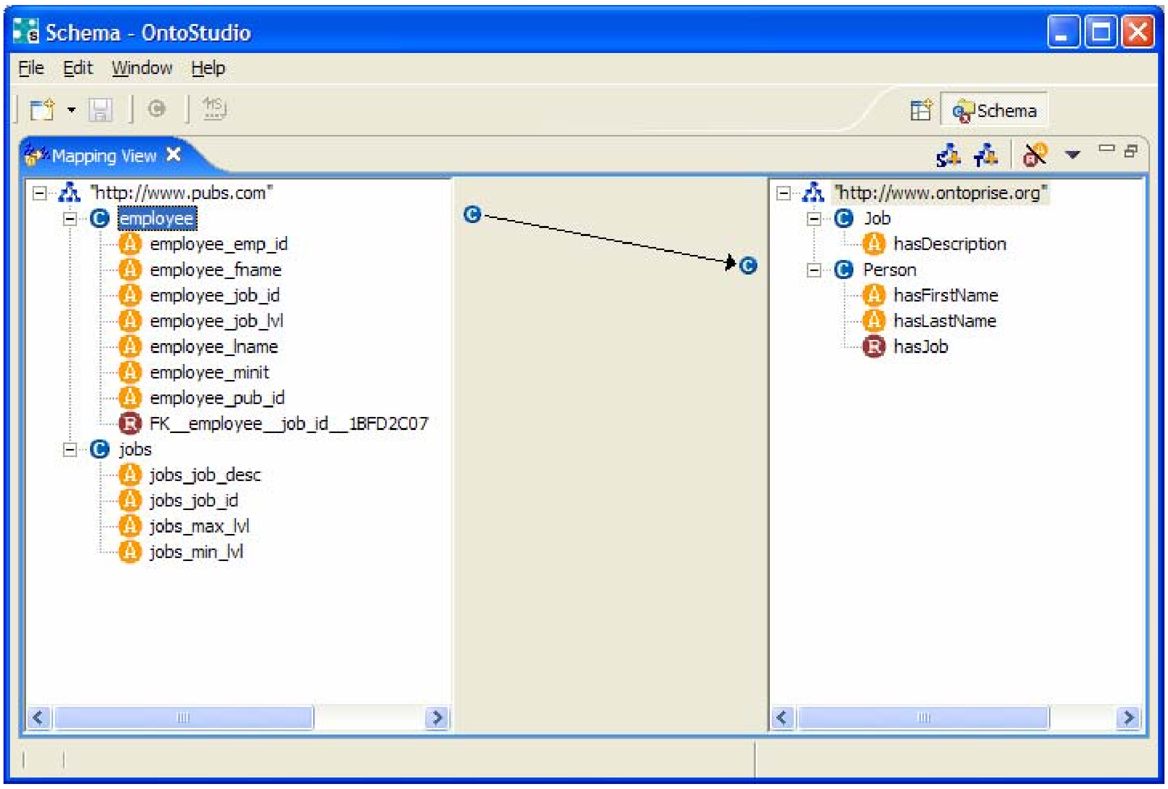

Figure 3: Mapping concepts on concepts

By dragging a concept from the left to a concept on the right a concept-to-concept-mapping is created. In our example shown in Figure 3 we map ‘employee’ from the flat source ontology to our enterprise ontology and by that make facts from the employee database accessible to our enterprise ontology.

2.2 Attributes on concepts

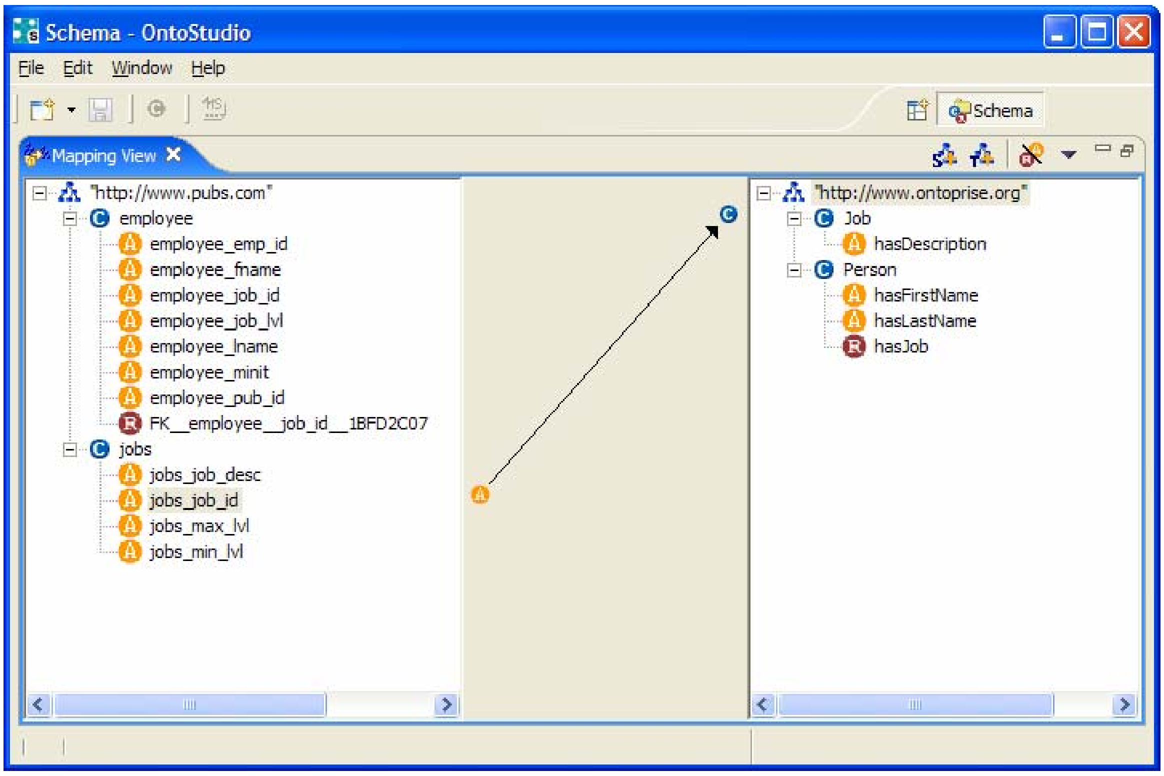

Figure 4: Mapping attributes on concepts

Draw a connection from an attribute on the left to a concept on the right to create an attribute-to-concept-mapping. In Figure 4 each attribute value of ‘jobs_job_id’ from the source will become an instance of ‘Job’ in the target ontology.

2.3 Attributes on attributes

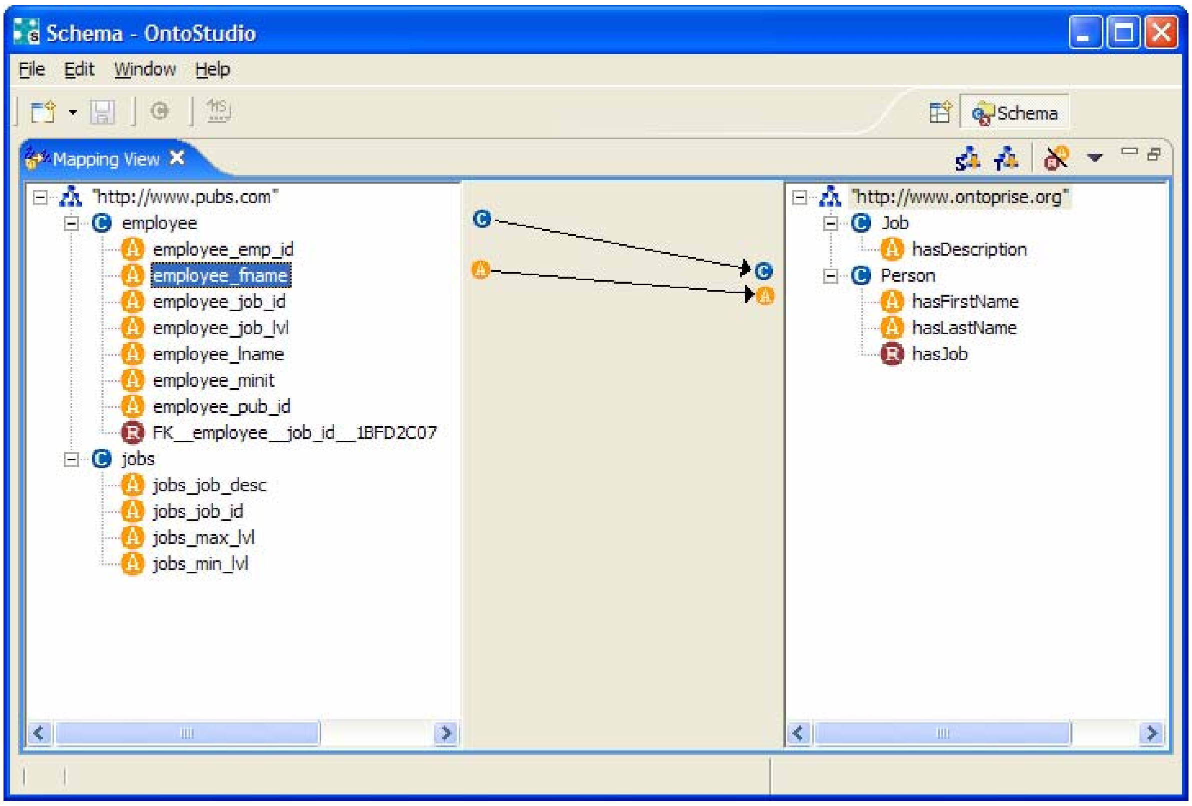

Figure 5: Mapping attributes on attributes

An attribute-to-attribute-mapping is defined by connecting two attributes. If there's no concept-to-concept- or attribute-to-concept-mapping their concepts are mapped automatically. When mapping the attributes 'employee_fname' to 'hasFirstName' in Figure 5 the concepts 'employee' and 'Person' will automatically be mapped if they weren't before.

2.4 Relations on relations

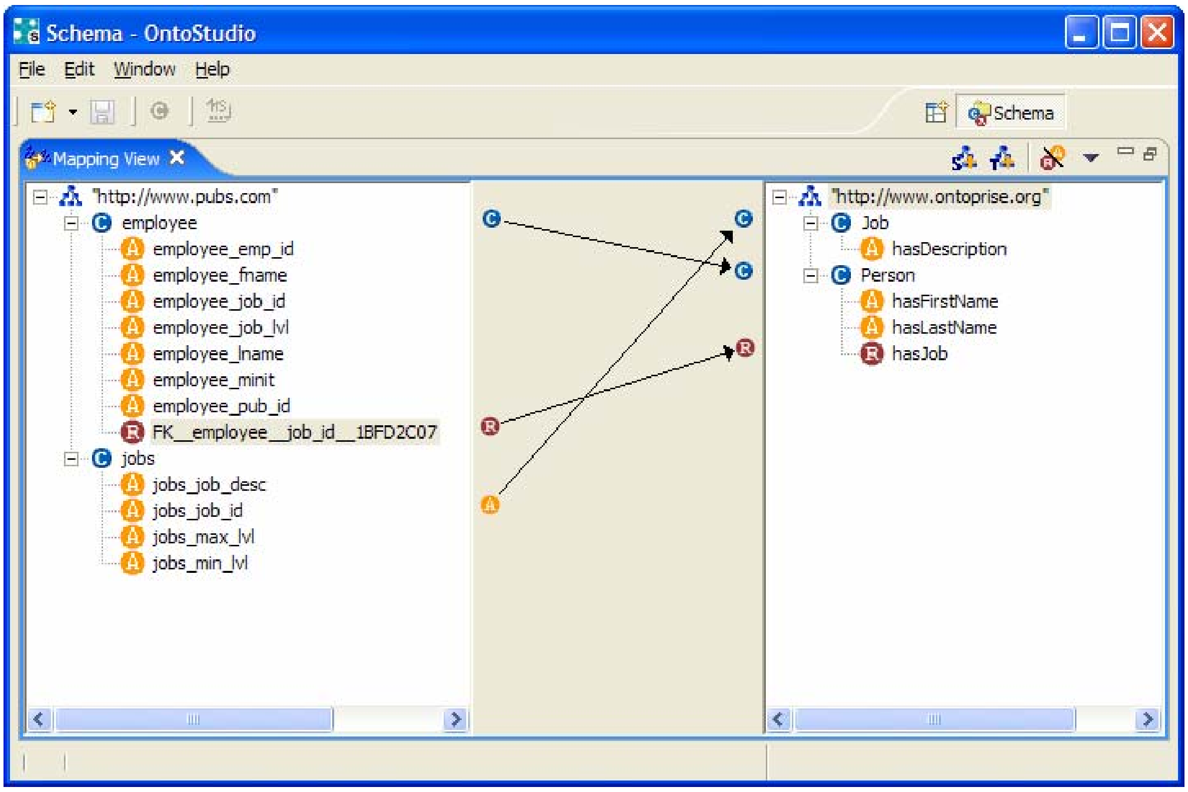

Figure 6: Mapping relations on relations

To map relations on relations all allocated concepts have to be mapped. When this is not done before OntoMap generates all required concept-to-concept-mappings automatically. When mapping the relation 'FK__employee__job_id__1BFD2C07' to 'hasJob' in Figure 6 it is presumed that the attribute-to-concept-mapping 'jobs_job_id' to 'Job' existed before, otherwise the concept-to-concept-mapping 'jobs' to 'Job' had been generated.

3 Integration of the different components

3.1 Mapping store – Mapping Component

To be able to access all the existing data, which is stored in the different repositories, connectors to the repositories will be developed for OntoStudio during this project. With these connectors we will be able to retrieve ontologies and mappings from the repositories as well as to store new defined mappings in the repository.

Let's take a look at a simple scenario. Having two ontologies, a domain specific ontology and a second ontology which shall be the integration ontology, we want to do some integration stuff.
In a first step we might use the mapping discovery described in WP4.4.1 to generate possible mappings. In a second step we now want to verify the correctness of the discovered mappings and to finalize the mappings by hand.

Therefore the user connects OntoStudio with the repositories to retrieve the ontologies and the mappings defined between them. In the Mapping View the already existing mappings are now shown and can be modified or deleted. Completely new mappings can also be defined.
If we have finished this process, the modified information will be stored back to the repository by the use of an export functionality.

OntoStudio will also provide an administration tool, to manage the different repositories and their access information.

3.2 Mapping patterns – Mapping Component

The mappings that can graphically be defined by the mapping tool in OntoStudio are based on the mapping patterns defined in D4.3.1.
The simple mapping patterns like e.g."concept to concept"-mappings can easiliy be defined by graphical means. Even more complex patterns can be represented in a graphical way. For example "attribute to attribute"-mappings with value conversions are much more complicated, but can be defined in a graphical way.
But the number of patterns that are expressible by an intuitive user interface is limited. So the mapping capabilities integrated in OntoStudio will be limited to this amount of patterns which are expressable by a real intuitive user interface.

4 Component Description

4.1 Data Model

4.1.1 Pattern

By "pattern" below we will assume template-patterns, not instantiated patterns. A pattern contains the following attributes:

- Definition – the definition of the pattern is specified in the terms of the pattern language, as given in the deliverable D.4.3.1.
- Name – name or title of the pattern, not necessarily unique
- Description - human-readable informal description of the pattern
- Related patterns – list of references to patterns with respect to *part-of*, *is-a* or *related-to* semantic relations.
- Pattern ID – in addition each template pattern is associated with pattern identifier, taken from the pattern in the Pattern Store (described below).

4.1.2 Mapping

The structure of the mapping object is determined by three important requirements:

1. Instantiated mapping patterns can be part of the definition of the mapping or even represent it at all. In particular each instantiated pattern is a mapping;
2. Each ontology mapping is bound to two ontologies;
3. Versioning should be supported.

As a result we get the following model specification:

- Definition – in terms of the mapping language, the specification of which will be provided in deliverable D.4.3.1.
- Name – mapping name/title, not necessarily unique
- Version – necessary for the mapping versioning. We intend to support version in the form MAJOR.MINOR.BUILD, e.g. "1.3.115". However, we do not want to enforce this, so the first version of the API will support arbitrary strings as well.
- Description – human specific information for the mapping
- Source and Target Ontologies – each mapping keeps references to the ontologies for which it is specified. The ontology objects are defined at java level in wsmo4j[2].
- Referenced patterns – a list of references to template patterns which have been used for the ontology mapping through instantiating.

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)
```

- [TBD]. Mapping ID - similarly to the patterns each ontology mapping is associated with mapping identifier got after storing the mapping in the Mapping Store (described below).

4.2 Store

4.2.1 Pattern and Mapping Store

We provide the interfaces PatternStore and MappingStore in order to allow storage and retrieval of patterns and mappings.. There could be multiple implementations for them, e.g. File-based, RDBMS-based or Full-text. The StorageFactory class is an implementation-hiding factory, used to create Patter/MappingStore instances.

To load objects from the store one needs an ID. The loading is preceded by a search through some kind of specific attribute restriction(s). We define the pattern- and the mapping restrictions bellow. Usually, a restriction returns more than one result, so a list of patterns/mappings can be expected.

Removal of patterns and mappings will also be supported. Because each mapping can have references to some patterns, the store will allow to remove a pattern only if there are not any references to it.

4.2.2 Pattern and Mapping Restrictions

We will consider a set of restrictions split in simple restrictions and composite restrictions, where simple restrictions are single restrictions on a single attrubute, while the composite restrictions consist of several simple ones.

Three kinds of simple restrictions are considered:

- StringRestriction – checks if given string starts with, ends with, matches to, contains or equals to a specified restriction string.
- ValueRestriction – checks if given comparable object is lower, upper or equal to a specified by the restriction comparable object.
- ListRestriction – checks if given object is present in a list of objects.

Each of the above simple restrictions uses only one of the mentioned conditions. For example, **StringRestriction** that **startsWith** "**X**" is a simple restriction, while **ValueRestriction(1<x<10)** is a composite restriction that is composed of two simple ones: **ValueRestriction(1<x)** and **ValueRestriction(x<10)**.

Composite restrictions also allow imposing restrictions on more than one attribute, for example:

- PatternRestriction – checks if a given pattern complies with the imposed restrictions to its components:
  Name – some StringRestrictions
  Description – some StringRestrictions
  Part-of related patterns – a ListRestriction
  Is-a related patterns – a ListRestriction
  Related-to related patterns – a ListRestriction
- MappingRestriction – checks if a given mapping complies with the imposed restrictions to its components:
  Name – some StringRestrictions
  Version – some ValueRestrictions
  Description – some StringRestrictions
  Source ontology – some StringRestrictions
  Target ontology – some StringRestrictions
  Referenced Patterns – some ListRestrictions

An example in java follows:

```
PatternRestriction patternRestr = new PatternRestriction();
StringRestriction nameRestr1 =
            new StringRestriction(StringRestriction.STARTS_WITH, "sub");
StringRestriction nameRestr2 =
            new StringRestriction(StringRestriction.CONTAINS, "By");
// suppose that list contains one or more PatternID objects.
```

```
   ListRestriction listResr = new ListRestriction(list);


   patternRestr.addRestriction(PatternRestriction.NAME_RESTRICTION,
nameRestr1);
   patternRestr.addRestriction(PatternRestriction.NAME_RESTRICTION,
nameRestr2);
   patternRestr.addRestriction(PatternRestriction.RELATED_TO_RESTRICTION,
listRestr);
```

4.3 Instance Transformation

The Instance Transformation component allows to transform instances or sets of instances between two ontologies, according to a given mapping. The ontologies are specified in the mapping itself (because each Mapping object is specified as a mapping between two ontologies). The two different transformations are called:

- “batch mode” – a set of instances (or all instances) will be transformed in a non-interactive mode;
- “run-time mode” - only one instance (and possibly the related ones) will be transformed.

An essential part of the process is the check if the instance being transformed already exists in the target ontology. This is done using the Instance Unification component (defined below), which answers whether two instances are equal. However, in the process of transforming instances, it is better to avoid invoking the unification component for each two instances (because the algorithm will become quadratic), especially if we expect large number of instances. Therefore we must take steps to prevent this. One important assumption is that only instances of compatible classes can be equal. By compatible classes we mean that one of the classes is a subclass of the other. It makes sense to impose even stronger restrictions on the instance types and the API for Instance Unification component allows for that.

For example, if after mapping transformation the result instance is of type **City,** it makes sense to consider all instances of type **City, LocalCapital, CountryCapital** for possible unification, but it is senseless to consider the instances of type **Person**.

4.4 Instance Unification

The Instance Unification component essentially answers whether two instances **in the target ontology** are the same. It is used in the last stage of the `InstanceTransformation`.

There are two additional requirements that contribute to the definintion the API for instance unification:

- As we mentioned above it’s possible to take some preliminary steps to reduce the number of all candidate instances for unification;
- As the last step we should merge the two instances in the target ontology.

It is impossible to do unification between instances in the general case. When we do unification we utilize knowledge related to the mapping, the ontology or the meaning of the specific instance types. With respect to this, we allow the specification if Instance Unification to be done in the following three ways:

- default unification – the implementation involves measures of the similarity and compatibility between instances. This is only possible when knowledge of the intended meaning of the concepts from the target ontology is available (for example, if the ontology is mapped to PROTON ontology, see D1.8.1 "Base Upper Level Ontology Guidance")
- Per-ontology instance unification – there could be ontologies, for which it makes sense to define specific instance unification. After that, all mappings having this ontology as target, could utilize the specified unification.
- Per-mapping – this is to allow the user to define specific unification for a particular mapping.

4.4.1 Cascading unifications

In addition to the three types of instance unification, one may want to cascade some of them. A simple example is if one wants to define a specific unification for one concept, and use the default unification for the rest.

Such cascading mechanism for unification of instances can be put into effect if

1. each "mapping unification" has reference to either the "default unification" or some compatible "ontology unification" (in sense of referring to the same target ontology)
2. each "ontology unification" has reference to the "default unification"

So if in some "mapping unification" it's not possible to give an answer whether two instances can be unified then we will expect the answer by the corresponding "ontology unification". As a last resort the "default unification" must reply to the question without hesitation. For example:

Let's consider a source ontology **EuropeGeography** and a target ontology **WorldGeography.**

Mapping unification – for each two instances i1 and i2 of the ontology WorldGeography

```
        if (i1 is subregion of Europe and i2 is not subregion of Europe)
        then ( the instances can't be unified)
        else ( call Ontology unification for i1 and i2)
```

Ontology unification – for each two instances i1 and i2 of the ontology WorldGeography

```
        if (i1 is City & i2 is City & i1.name == i2.name & i1.population == i2. population)
        then (the instances refer to the same real object)
        else (call Default unification for i1 and i1)
```

Default unification – for each two instances i1 and i2 of the ontology WorldGeography answer that they are not the same.

5 API

The API section defines the objects at java-level and is organized as follows:

- Overview/notes
- Notes on creation of the objects (including constructors)
- List of public methods, with notes and discussion
- Example usage

5.1 Data Model

Here follows list of constructors and methods for Pattern and Mapping objects given in Java-Doc like notation in alphabetical order.

5.1.1 Pattern

- Each pattern can be created in two ways:
  1. By importing the pattern definition from a specific language format through parser object.
  2. By loading the pattern from the Pattern Store.

Constructor Summary
`Pattern(Reader inputDefinition, Parser parser)` Constructs a pattern with definition imported from a specific language format through given parser. The initial format is retrieved by given input stream.

- The encapsulation of the pattern object fields is performed via standard get/set methods:

Method Summary	
`String`	`getDefinition()`
`String`	`getDescription()`
`Iterator`	`getIsAPatterns()`
`String`	`getName()`
`Iterator`	`getPartOfPatterns()`
`Iterator`	`getRelatedToPatterns()`
`Void`	`setDescription(String desc)`
`void`	`setIsAPatterns(List partOfPatterns)`
`void`	`setName(String name`
`void`	`setPartOfPatterns(List partOfPatterns)`
`void`	`setRelatedToPatterns(List partOfPatterns)`

5.1.2 *Mapping*

- Each mapping can be created in two ways:
  1. By importing the mapping definition from a specific language format through given parser object.
  2. By loading a mapping from the Mapping Store.

Constructor Summary
Mapping(Reader inputDefinition, Parser parser, Ontology sourceOntology, Ontology targetOntology) Constructs a mapping for two given ontologies with definition imported from a specific language format through given parser. The initial format is retrieved by given input stream.

- The encapsulation of the mapping object fields is performed via standard get/set methods.

Method Summary	
String	**getDefinition**()
String	**getDescription**
String	**getName**()
List	**getReferencedPatterns**()
Ontology	**getSourceOntology**()
Ontology	**getTargetOntology**()
Version	**getVersion**()
void	**setDescription**(String descr)
void	**setName**(String name)
void	**setReferencedPatterns**(List patterns)
void	**setVersion**(Version v)

5.1.3 *Version*

- Version has three attribute fields. Only the first one referring to the major version must be always initialized. Whereas it's not necessary the other ones to be represented.

Field Summary	
String	**buildNumber**
String	**major**
String	**minor**

- Each version can be created in four ways, but two of them are essentially different:
  1. By standard initializing each one of the class attributes.

2. By string that must be split to parts through given (or default) delimiter. Each of them refers to some of the attributes respectively.

Constructor Summary
Version(String sVersion) Constructs a new Version from string splited through default delimiter. See the constructor Version(String, char).

Version(String sVersion, char delimiter)
Constructs a new Version from string splited through default delimiter. For Example if delimiter is '.' and:
sVersion = "Alpha.1.5" => `Major-Version` = "Alpha", `Minor-Version` = "1", `BuildNumber` = "5"
sVersion = "5.1" => `Major-Version` = "5", `Minor-Version` = "1"
sVersion = "Version3" => `Major-Version` = "Version3"

Version(String major, String minor) Constructs a new Version with given major and minor values, without buildNumber component.
Version(String major, String minor, String buildNumber) Constructs a new Version with given major, minor and buildNumber values.

- The fields of version object are encapsulated via standard get/set methods. Also Version is implementation class of Comparable interface and corresponding compareTo and equals methods should be implemented. To notice that we consider two versions as comparable only if they are represented by the same list of components.

Method Summary	
int	**compareTo**(Object v) This method is specified by Comparable interface. To notice that we consider two versions as comparable only if they are represented by the same list of components. In opposite case ClassCastException will be thrown.
boolean	**equals**(Object v) Compares this version to the specified object. The result is `true` if and only if compareTo method applied to this object returns 0.
String	**getBuildNumber**()
String	**getMajorVersion**()
String	**getMinorVersion**
void	**setBuildNumber**(String s
void	**setMajorVersion**(String s)
void	**setMinorVersion**(String s
String	**toString**()

5.2 Store

5.2.1 Pattern Store

- There could be multiple implementation classes for this interface, e.g. File-based, Database, Full-text. A reference to Pattern Store object could be obtained from Storage Factory
- The following methods allow storage, search and retrieval of patterns.

Method Summary	
`Pattern`	**`loadPattern`**`(PatternID pattern)` Loads a pattern through its identifier
`void`	**`removePattern`**`(PatternID pattern)` Removes a pattern specified by its identifier.
`Iterator`	**`searchByIsARelation`**`(Pattern pattern)` Searches for all patterns which have relation of type is-a with a given pattern
`Iterator`	**`searchByKeywords`**`(List keywords)` Searches for all patterns with given keywords in their descriptions
`Iterator`	**`searchByName`**`(String patternName)` Searches for a pattern with given name
`Iterator`	**`searchByPartOfRelation`**`(Pattern pattern)` Searches for all patterns which have relation of type part-of with a given pattern
`Iterator`	**`searchByRelatedToRelation`**`(Pattern pattern)` Searches for all patterns which have relation of type related-to with a given pattern
`Iterator`	**`searchByRestriction`**`(PatternRestriction patternRestriction)` Searches for patterns in respect to given composite restriction
`PatternID`	**`storePattern`**`(Pattern pattern)` Stores a pattern and returns newly created pattern identifier

5.2.2 Mapping Store

- There could be multiple implementation classes for this interface, e.g. File-based, Database, Full-text. A reference to Mapping Store object could be obtained from Storage Factory
- The following methods allow storage, search and retrieval of mappings.

Method Summary	
`Mapping`	**`loadMapping`**`(MappingID id)` Loads a mapping through its identifier
`void`	**`removeMapping`**`(MappingID mapping)` Removes a mapping specified by its identifier.
`Iterator`	**`searchByKeywords`**`(List keywords)` Searches for all mappings with the given keywords in their descriptions
`Iterator`	**`searchByName`**`(String name)` Searches for an ontology mapping with given name
`Iterator`	**`searchByPatterns`**`(List patterns)` Searches for ontology mappings in respect to their list of related patterns.
`Iterator`	**`searchByRestriction`**`(MappingRestriction mappingRestriction)` Searches for ontology mappings in respect to given composite restriction
`Iterator`	**`searchBySourceOntology`**`(Identifier sourceOnt)` Searches for ontology mappings with specified source ontology
`Iterator`	**`searchByTargetOntology`**`(Identifier targetOnt)` Searches for ontology mappings with specified target ontology
`Iterator`	**`searchByVersion`**`(Version v)` Searches for an ontology mapping with given version
`MappingID`	**`storeMapping`**`(Mapping mapping)` Stores a mapping and returns new created mapping identifier

5.2.3 Storage Factory

Method Summary	
`static MappingStore`	**`getMappingStore`**
`static PatternStore`	**`getPatternStore`**`()`

5.2.4 Pattern/Mapping Restriction

Pattern and Mapping Restriction classes implement the Composite Restriction interface, which extends Restriction interface.

5.2.4.1 Restriction

Restriction is an interface with only one method.

Method Summary	
boolean	**admit**(Object o) Checks if given object complys with restrictions specified by the implementation class

In respect to the data types of the considered objects, three direct implementation classes are supported:

5.2.4.2 StringRestriction

Restricts string object

Field Summary	
Static int	**CONTAINS**
Static int	**ENDS_WITH**
Static int	**EQUALS**
Static int	**MATCHES**
Static int	**STARTS_WITH**

Constructor Summary
StringRestriction(int typeOfRestriction, String s) Constructs basic restriction by string and type. The range of the first argument is set of values of class variables referring to the type of the restriction.

5.2.4.3 ValueRestriction

Restricts comparable object

Field Summary	
static int	**EQUAL**
static int	**LOWER**
static int	**UPPER**

Constructor Summary
ValueRestriction(int typeOfRestriction, Comparable value) Constructs basic restriction by comparable object and type. The range of the first argument is set of values of class variables referring to the type of the restriction.

5.2.4.4 ListRestriction

Restricts object in respect to this if it is presented in given list of objects or not.

Constructor Summary
`ListRestriction(List contains)` Constructs basic restriction by list of objects.

5.2.4.5 CompositeRestriction

CompositeRestriction is an interface with one method extending RestrictionInterface.

Method Summary	
`void`	`addRestriction(int restrictedField, Restriction r)` This method add restriction to some attribute of the implemantation class. The range of the first argument is set of values of class variables referring to the class attributes.

Each implementation class of CompositeRestriction interface enables to be added one or more "simple" restrictions that refer to the class attributes. As we said before pattern and mapping restrictions are such examples:

5.2.4.6 PatternRestriction

Restricts object of type Pattern

Field Summary	
`static int`	`DESCRIPTION_RESTRICTION`
`static int`	`IS_A_RESTRICTION`
`static int`	`NAME_RESTRICTION`
`static int`	`PART_OF_RESTRICTION`
`static int`	`RELATED_TO_RESTRICTION`

Constructor Summary
`PatternRestriction()`

Method Summary	
`void`	**addRestriction**`(int typeOfRestriction, Restriction r)` This method is specified by CompositeRestriction Interface. The range of the first argument is set of values of class variables referring to the class attributes.
`boolean`	**admit**`(Object o)` This method is specified by Restriction Interface. It checks if given pattern complies with all those restrictions which have been included through addRestriction method or it doesn't.

5.2.4.7 MappingRestriction

Restricts object of type Mapping.

Field Summary	
`static int`	**DESCRIPTION_RESTRICTION**
`static int`	**NAME_RESTRICTION**
`static int`	**PATTERN_RESTRICTION**
`static int`	**SOURCE_ONTOLOGY_RESTRICTION**
`static int`	**TARGET_ONTOLOGY_RESTRICTION**
`static int`	**VERSION_RESTRICTION**

Constructor Summary
MappingRestriction`()` Empty constructor.

Method Summary	
`void`	**addRestriction**`(int typeOfRestriction, Restriction r)` This method is specified by CompositeRestriction Interface. The range of the first argument is set of values of class variables referring to the class attributes.
`boolean`	**admit**`(Object o)` This method is specified by Restriction Interface. It checks if given mapping complies with all those restrictions which have been included through addRestriction method or it doesn't.

5.2.5 Example Scenarios

This section contains some simple scenarios illustrating the functionality of the Pattern and Mapping Store components.

5.2.5.1 Pattern Store Scenarios

The first thing is to create a Pattern. This is preformed via pattern definition and Parser object:

```
String patternDef  = "equivalentClassMapping(C1 C2)";
Reader r = new StringReader(patternDef);
Pattern examplePattern = new Pattern(r, parser);
String patternName = "EquivalentClassMapping";
String patternDescr = "A class in one ontology has the same meaning as a class in a second ontology";

examplePattern.setName(patternName);
examplePattern.setDescription(patternDescr);
```

Afterwards, a reference to Pattern Store should be retrieved from the Storage Factory to allow storing of the new created pattern:

```
PatternStore ps = StorageFactory.getPatternStore();
PatternID id = ps.storePattern(examplePattern);
```

After this if we want to find this pattern and modify it, we will create a pattern restriction respectively:

```
PatternRestriction patternRestr = new PatternRestriction();
StringRestriction nameRestr =
   new StringRestriction(StringRestriction.STARTS_WITH, "equivalent");


patternRestr.addRestriction(PatternRestriction.NAME_RESTRICTION, nameRestr);
Iterator equivPatterns = ps.searchByRestrictions(patternRestr);

PatternID patternID;
Pattern curPattern = null;
for(int i = 0; equivPatterns.hasNext(); i++) {
    patternID = (PatternID) (equivPatterns.next());
    curPattern = ps.loadPattern(patternID);
    System.out.println(" " + patternID.toString() + ": " + curPattern.getDescription());
}
```

Finally we will modify the description of the first pattern complying with the specified restriction

```
Iterator listPatterns = ps.searchByRestrictions(patternRestr);
PatternID firstPatternID;
Pattern firstPattern;

if(listPatterns.hasNext()) {
    firstPatternID = (PatternID)(listPatterns.next());
    firstPattern = ps.loadPattern(firstPatternID);
    firstPattern.setDescription("A class in the one ontology has the same meaning as a class in the second ontology. This is a common pattern");
}
```

5.2.5.2 Mapping Store Scenarios

Firstly we will create a simple mapping between two given ontologies “Philosophy” and “Psychology”. For this it’s necessary a mapping definition and a reference to a Parser object

```
String mappingDef = "classMapping(o1:Human o2:Person" +
                    " attributeMapping(o1:name o2:name)" +
                    " attributeMapping(o1:age o2:age))";
Ontology sourceOnt;  // reference to WSMO Ontology concerning to o1
Ontology targetOnt;  // reference to WSMO Ontology concerning to o2

Reader r = new StringReader(mappingDef);
Mapping exampleMapping = new Mapping(r, parser, sourceOnt, targetOnt);


String mappingName = "Philosophy2Psychology";
String mappingDescr = "Searching the relationships between humanity and personality.";
Version initialVersion = new Version("Beta.1");

exampleMapping.setName(mappingName);
exampleMapping.setDescription(mappingDescr);
exampleMapping.setVersion(initialVersion);
```

Afterwards, a reference to Mapping Store should be retrieved from the Storage Factory to allow storing of the new created mapping:

```
MappingStore ms = StorageFactory.getMappingStore();
MappingID id = ms.storeMapping(exampleMapping);
```

After this if we want to find this mapping and modify it, we must preliminarily create a mapping restriction:

```
    MappingRestriction mappingRestr = new MappingRestriction();
    ArrayList listOfKeyWords = new ArrayList(2);
    listOfKeyWords.add("humanity");
    listOfKeyWords.add("personality");
    ListRestriction descrRestr = new ListRestriction(listOfKeyWords);
    StringRestriction nameRestr =
       new StringRestriction(StringRestriction.CONTAINS, "Psychology");


    mappingRestr.addRestriction(MappingRestriction.DESCRIPTION_RESTRICTION, descrRestr);

    mappingRestr.addRestriction(MappingRestriction.NAME_RESTRICTION, nameRestr);

    Iterator mappings = ms.searchByRestrictions(mappingRestr);

    MappingID mappingID;
    Mapping curMapping;
    while(mappings.hasNext()) {
        mappingID = (MappingID)(mappings.next());
        curMapping = ms.loadMapping(mappingID);
        System.out.println(" " + mappingID.toString() + ": " + curMapping.getDescription());
    }
```

Finally, we will edit some fields of the mapping. After that a new version of it will be created allowing the old version to be preserved.

```
    MappingID firstMappingID;
    Mapping firstMapping;
    Version mappingVersion;
    String newMinorVersion;

    Iterator mappingList = ms.searchByRestrictions(mappingRestr);
    if(mappingList.hasNext()) { // edit only first mapping in the list
        firstMappingID = (MappingID)(mappings.next());
        firstMapping = ms.loadMapping(firstMappingID);
        firstMapping.setDescription("Mapping between Philosophy Ontology and Psychology Ontology finding out the relationships between humanity and personality.");
```

```
            mappingVersion = firstMapping.getVersion();
            newMinorVersion = "" +
(Integer.parseInt(mappingVersion.getMinorVersion()) + 1);
            mappingVersion.setMinorVersion(newMinorVersion);
            firstMapping.setVersion(mappingVersion);
        }
```

5.3 Instance Transformation

Instance Transformation is an interface with two methods for “batch mode” and “run-time mode” respectively. The instance objects are defined at java level in wsmo4j.

Method Summary	
`Collection`	**`transformation`**`(Mapping mapping, Collection instancies, InstanceUnification iu)` Transforms a set of instances of one ontology to the corresponding instances of second ontology according to a given mapping. The InstanceUnification parameter is the particular instance unification that has to be used when transforming instances.
`Instance`	**`transformation`**`(Mapping mapping, Instance i, InstanceUnification iu)` Transforms an instance of one ontology to an instance of second ontology according to a given mapping. The instance unification parameter is the same as in the above method.

5.4 Instance Unification

InstanceUnification is an interface with possible different implementations depending on the point of view – mapping, ontology or by default. The instance and ontology objects are defined at java level in wsmo4j.

The constructor should take “unification parent” parameter used to implement cascading.

Method Summary	
`Ontology`	**`getTargetOntology`**`()` Gives a reference to the target ontology for which this instance unification is aplicable
`Int`	**`isTheSame`**`(Instance i1, Instance i2)` Checks whether two instances of the target ontology are the same. Returns the value 0 if it’s not possible to give an answer, the value 1 if they refer to the same real object or -1 otherwise.

Instance	merge(Instance i1, Instance i2) Unifies two instances of the target ontology creating new instance on the place of the first one. It's possible to make unification only if isTheSame() method returns 1 and so a reference to the new created instance(first one) will be returned.

6 Bibliography and references

[1] SEKT, D4.4.1 Ontology Mediation Management
[2] WSMO4J, (http://wsmo4j.sourceforge.net/)