D4.3.2 Ontology Mediation Patterns Library V2

François Scharffe (DERI Innsbruck)
Jos de Bruijn (DERI Innsbruck)
Douglas Foxvog (DERI Galway)

Abstract.
EU-IST Integrated Project (IP) IST-2003-506826 SEKT
Deliverable D4.3.2 (WP4)
This deliverable describes a library of ontology mapping patterns, as well as a mapping language based on these patterns. This language, together with the mapping patterns, allows the user to more easily identify mappings and to describe mappings in an intuitive way. The mappings are organized in a library in a hierarchical fashion in order to allow for easy browsing and retrieving of mappings.
Keyword list: Ontology Mapping, Mapping Patterns, Patterns Library

Document Id.	SEKT/2005/D4.3.2/v2.0
Project	SEKT EU-IST-2003-506826
Date	February 4, 2006
Distribution	public

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

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 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

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

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

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

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

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 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

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

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

Universitat Autonoma de Barcelona
Edifici B, Campus de la UAB
08193 Bellaterra (Cerdanyola del Vallès)
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

Contents

Chapter 1

Introduction

Implementing is often seen as the bad task, given to the programmer as the handworker to build the house designed by the software architect. The specification of a language is nothing if no implemented tools are following it. In this second version of the patterns library deliverable we present an update of the mapping language and pattern library presented in the first version. We add a RDF syntax to the mapping language, grounding to different ontology languages, as well as a study of instances transformation functions. Most of the work in the task corresponding to this deliverable was implementation of the tools necessary to the use of mapping as we envision it in the Sekt project. These tool are presented in section 5.

Ontology mappings can be used for different tasks, such as data transformation and ontology merging (cf. D4.4.1 [8]). In order to enable automation in these tasks, ontology mappings must specify the relationship between different ontologies in a formal way. As such, it is possible to view an ontology mapping as a collection of logical formulae. Notice, however, that logical formulae are in general very hard to understand and even harder to model correctly. Thus, it is beneficial to provide guidance in the mapping task in the form of a human-understandable mapping language and in the form of recurring patterns of ontology mapping. This language must still remain machine readable to be at the basis of user-friendly tools like a graphical mapping editing tool.

This section is further structured as follows. We first clarify the terminology used in this deliverable in Section 1.1. In order to understand the relation between mapping patterns and the mapping language developed in this deliverable, we explain the relation between the mapping language and the mapping patterns in Section 1.2.

1.1 Terminology

In order to make this deliverable self-contained we present here a slightly adapted version of the terminology clarification we have provided earlier in deliverable D4.2.1 [7].

This section provides some clarification on the terminology used throughout this deliverable. We deem this necessary, because there exist many different understandings of the terminology in the literature.

Ontology An *ontology* O is a 4-tuple $\langle C, R, I, A\rangle$, where C is a set of concepts, R is a set of relations, I is a set of instances and A is a set of axioms. Note that these four sets are not necessarily disjoint (e.g. the same term can denote both a class and an instance), although the ontology language might require this. Each concept can have a number of *attributes* associated with it. An attribute is a special kind of relation, namely a binary relation associated with a concept.

All concepts, relations, instances and axioms are specified in some logical language. This notion of an ontology coincides with the notion of an ontology described in [28, Section 2] and is similar to the notion of an ontology in OKBC [4]. Concepts correspond with classes in OKBC, slots in OKBC are particular kinds of relations, facets in OKBC are a kind of axiom and individuals in OKBC are what we call *instances*[1].

In an ontology, concepts are usually organized in a subclass hierarchy, through the *is-a* (or subconcept-of) relationship. More general concepts reside higher in the hierarchy.

Instance Base Although instances are logically part of an ontology, it is often useful to separate between *an ontology* describing a collection of instances and *the collection of instances* described by the ontology. We refer to this collection of instances as the *Instance Base*. Instance bases are sometimes used to discover similarities between concepts in different ontologies (e.g. [32], [12]). An instance base can be any collection of data, such as a relational database or a collection of web pages. Note that this does not rule out the situation where instances use several ontologies for their description.

Instances are an integral part of an ontology. However, we expect that most instance data will be stored in private data stores and will not be shared along with the ontology. The instances contained in the ontology itself are typically those instances that are shared.

Note that in a Semantic Web setting, each instance is identified with a URI (or IRI). Furthermore, a particular individual can be an instance of multiple concepts which might belong to different ontologies. Since an instance base belongs to one ontology, an instance can belong to multiple instance base.

Ontology Language The ontology language is the language which is used to represent the ontology. Semantic Web ontology languages can be split up into two parts: the logical and the extra-logical parts. The *logical* part amounts to a theory in some

[1]We use the terms instance and individual interchangeably throughout this document. Note that an instance is not necessarily related to a class.

logical language, which can be used for reasoning. Class (concept) definitions, property (relation) definitions, and instance definitions correspond with axioms in the logical language. In fact, such definitions are merely a more convenient way to write down such axioms.

The *extra-logical* part of the language typically consists of non-functional properties (e.g. author name, creation date, natural language comments, multi-lingual labels; see also Dublin Core [35]) and other extra-logical statements, such as namespace declarations, ontology imports, versioning, etc.

Non-functional properties (also called *annotations*) are typically only for the human reader, whereas many of the other extra-logical statements are machine-processable. For example, namespace declarations can be used to resolve Qualified Names to full URIs and the importing of ontologies can be achieved automatically by either (a) appending the logical part of the imported ontology to the logical part of the importing ontology to create one logical theory or (b) using a *mediator*, which resolves the heterogeneity between the two ontologies (see also the definition of **Ontology Mediation** below).

Ontology Mediation Ontology mediation is the process of reconciling differences between heterogeneous ontologies in order to achieve inter-operation between data sources annotated with and applications using these ontologies. This includes the discovery and specification of *ontology mappings*, as well as the use of these mappings for certain tasks, such as query rewriting and instance transformation. Furthermore, the *merging of ontologies* also falls under the term ontology mediation.

Ontology Mapping An *ontology mapping* M is a (declarative) specification of the semantic overlap between two ontologies O_S and O_T. This mapping can be one-way (injective) or two-way (bijective). In an injective mapping we specify how to express terms in O_T using terms from O_S in a way that is not easily invertible. A bijective mapping works both ways, i.e. a term in O_T is expressed using terms of O_S and the other way around.

Note that an ontology mapping is often *partial*, which means that the mapping does not specify the complete semantic overlap between two ontologies, but rather just a part of this overlap which is relevant for the mapping application.

Mapping Language The mapping language is the language used to represent the *ontology mapping* M. It is important here to distinguish between a specification of the similarities of entities between ontologies and an actual ontology mapping. The specification of similarities between ontologies is usually a level of confidence (usually between 0 and 1) of the similarity of entities, whereas an ontology mapping actually specifies the relationship between the entities in the ontologies. This is typically an exact specification and typically far more powerful than simple similarity measures. Mapping languages often allow arbitrary transformation between ontologies, often using a rule-based formalism and typically allowing arbitrary value transformations, as well as renaming and structural transformations.

Mapping Pattern Although not often used in current approaches to ontology mediation, patterns can play an important role in the specification of ontology mappings, because they have the potential to make mappings more concise, better understandable and reduce the number of errors (cf. [24]). A *mapping pattern* can be seen as a template for mappings which occur very often. Patterns can range from very simple (e.g. a mapping between a concept and a relation) to very complex, in which case the pattern captures comprehensive substructures of the ontologies, which are related in a certain way.

Mapping patterns are furthermore useful for graphical ontology mapping tools; mappings could be treated different in the user interface.

Matching We define *ontology matching* (sometime also called *mapping discovery*) as the process of discovering similarities between two source ontologies. The result of a matching operation is a specification of similarities between two ontologies. Ontology matching is done through application of the *Match* operator (cf. [27]). Any schema matching or ontology matching algorithm can be used to implement the *Match* operator, e.g. [12, 17, 20, 22].

We adopt here the definition of *Match* given in [27]: "[*Match* is an operation], which takes two schemas [or ontologies] as input and produces a mapping between elements of the two schemas that correspond semantically to each other".

For the definitions of merging, aligning and relating ontologies, we adopt the definitions given in [11]:

Ontology Merging Creating one new ontology from two or more ontologies. In this case, the new ontology will unify and replace the original ontologies. This often requires considerable adaptation and extension.

Note that this definition does not say how the merged ontology relates to the original ontologies. The most prominent approaches are the *union* and the *intersection* approaches. In the union approach, the merged ontology is the union of all entities in both source ontologies, where differences in representation of similar concepts have been resolved. In the intersection approach, the merged ontology consists only of the parts of the source ontology which overlap (c.f. the *intersection* operator in ontology algebra [36]).

Ontology Aligning Bringing the ontologies into mutual agreement. The ontologies are kept separate, but at least one of the original ontologies is adapted such that the conceptualization and the vocabulary match in overlapping parts of the ontologies. However, the ontologies might describe different parts of the domain in different levels of detail.

Relating Ontologies Specifying how the concepts in the different ontologies are related in a logical sense, i.e. creating an **Ontology Mapping**. This means that the original ontologies have not changed, but that additional axioms describe the relationship between the concepts. Leaving the original ontologies unchanged often implies that only a part of the integration can be done, because major differences may require adaptation of the ontologies.

The term "Ontology Mapping" was defined above as a specification of the relationship between two ontologies. We can also interpret the word "Mapping" as a verb, i.e. the action of *creating* a mapping. In this case the term corresponds with the term "Relating Ontologies":

Mapping Ontologies Is the same as relating ontologies, as specified above.

Note that most disagreement in the literature is around the term *alignment*. We do not use the term alignment as such, but we do use the term *ontology aligning*. In most literature (e.g. [23]), alignment corresponds with what we call *relating ontologies* or *mapping ontologies*. Ontology aligning is also sometimes called *ontology reconciliation*.

1.2 Mapping Language and Mapping Patterns

The mapping language and mapping patterns described in this deliverable are mutually dependant. In this section we clarify the relation between the ontology mapping language and the ontology mapping patterns.

This section is further structured as follows. We first outline some general considerations in the development of a mapping language and mapping patterns. We then describe the relationship between the mapping language and the mapping patterns, after which we describe the relationship between the mapping language and the actual mappings which are specified using the language.

1.2.1 General Considerations

Language-independent ontology mapping One of the goals of the mapping language is to capture general ontology mappings, independent of the particular ontology language. Unfortunately, this is not always possible, because of the differences in expressiveness and differences in modeling styles between ontology languages [9]. Our mapping patterns have a bias towards the ontology languages WSML-Flight [6] and OWL DL [10].

Although WSML-Flight and OWL have certain similarities, there are still major differences, which cannot be easily overcome. Since it is the goal of this deliverable to capture ontology mapping patterns and not to give a formal grounding for the mappings, we leave the formal grounding of the mappings to the SEKT deliverable D4.4.1 [8] and the DIP deliverable D1.5 [26], which ground the mapping language to OWL DL and WSML-Flight, respectively. Because the types of formulas which can be written down in different language have significant differences between the languages, we leave part of the syntax open in order to allow for language-specific extensions.

Ontology language/meta-model We see an ontology as a 4-tuple $\langle C, R, I, A\rangle$ with classes C, relations R, instances I and axioms A. Therefore, we group the elementary mapping patterns according to these four categories. Furthermore, a concept can have a number of attributes associated with it. An attribute is a special kind of relation, namely, a binary relation with a defined domain.

Mapping pattern template For the description of the individual mappings we develop a template in Chapter 4.1.

1.2.2 Relation between Mapping Patterns and the Mapping Language

The mapping patterns presented in Section 4.2 of this deliverable correspond with types of mappings which are expected to be encountered often in the practice of ontology mapping. These mapping patterns are very useful in guiding the developer of the ontology mapping to correctly construct ontology mappings. The mapping patterns can be used in a visual tool which is used for the specification of ontology mappings. Finally, the mapping patterns can be used as a guide for developers of ontology matching algorithms. A mapping pattern corresponds with a type of mapping that can be discovered using such an algorithm.

The mapping language described in this deliverable (see Chapter 3) is derived from the mapping patterns. However, we have chosen not to create a construct in the mapping language for each specific mapping pattern. Instead, the mapping constructs are based on the most general mapping patterns; additional constructs are introduced to create mappings which correspond to the more specific mapping patterns, in order to keep the language concise and understandable. A mapping pattern corresponds with an expression in the mapping language. For an overview of the correspondence between the mapping patterns and constructs in the mapping language see Table 4.3 of Section 4.3.

1.2.3 Relation between Mapping Language and actual mappings

In this deliverable we define only a reference semantics for the mapping language (see Appendix C). However, we do not require particular users of the mapping language to adhere to this semantics, because the actual semantics of the mappings depends on the semantics of the ontology language and requiring a particular semantics for the mapping language would decrease usability across different ontology languages. Therefore, it is not clear what a mapping specified using this language really means and it is not possible to execute any tasks with it, because the machine cannot interpret the statements written down using the language. Nonetheless, the conceptual correspondences between elements of the ontologies are captured by the language.

We believe that it is a good thing that the mapping language does not prescribe a particular semantics, because this means that the language can potentially be used for several different ontology languages.

For the formal semantics and the use of the mapping language for mapping ontologies specified using WSML and OWL, respectively, we refer the reader to the annexes B and A.

1.3 Implementation

The design of the mapping language and patterns allows us to specify mappings between ontologies having a common format. To enable the use of this technologies we must provide a set of tools. Later in this document we present the tools developed in the context of this project that allow effectively use the Sekt mapping technologies and apply them to the case studies of the project.

More specifically we will detail the two main components developed. The Mapping API is a Java implementation of tools giving the possibility to use the mapping language and the mapping patterns. This API includes several functionalities to parse the mapping patterns and mapping language constructs into an object model following the language structure. It also includes different serialization methods to ground the mappings to a specific ontology language. This grounding permitting to execute the mappings at run-time, using a reasoner like KAON2. We present these basic functionalities together with extended one chapter 5. This deliverable is about an ontology mapping pattern library. A hierarchy of mapping patterns is given but a tool implementing it is needed. In the same chapter we present the mapping store, giving the possibility to store mapping patterns and actual mappings as books are stored in a library. The mapping store is implemented as a Java API giving functionalities to store, search and retrieve the mappings based on different criterias.

This report is further structured as follows. Chapter 2 contains a number of motivating examples for the mapping patterns and the mapping language. Chapter 3 develops the ontology mapping language, based on the mapping patterns. Chapter 4 describes the mapping patterns identified in this deliverable. Chapter 5 describes the specific tools developed to deal with mappings and mapping patterns. Finally, we present conclusions in Chapter 6.

Chapter 2

Mapping Examples

In this chapter we present a number of example mapping scenarios which help to demonstrate the need for the mapping patterns and which provide a motivation for the development of the mapping language and the mapping patterns.

2.1 Motivating Mapping Scenarios

We present a number of ontology mapping scenarios which motivate particular aspects of ontology mapping. These mapping scenarios are taken into account in the development of the mapping language.

In the examples of this section we use F-Logic [19] notation because of its relatively concise and frame-based syntax. In short, F-Logic allows all of traditional predicate logic, i.e. function symbols $f, g, h, \ldots$, predicates $p, q, r, \ldots$, connectives $\wedge, \vee, \leftarrow, \leftrightarrow, \neg$, and quantifiers $\forall, \exists$. Notice that a nullary function symbols corresponds with a constant and a nullary predicate symbol corresponds with a proposition. In the examples, variables which are not explicitly quantified are implicitly universally quantified.

Additionally, F-Logic allows the following constructs: $A : B$ means that A is a member of class B; $A :: B$ means that A is a subclass of B, and $A[B \Rightarrow\!\!\!\!\!\!\Rightarrow C]$ means that A has an attribute B with value C. Furthermore, we also allow the symbol naf for negation-as-failure.

2.1.1 Join mappings

Suppose two classes in ontology O_1 are related to one class in O_2. In this case, it is common to either map the union or the intersection (depending on the relation between the classes) of the classes in O_1 to the class in O_2, in this case in a unidirectional mapping:

$x : C \leftarrow x : A \wedge x : B$

and

$x : C \leftarrow x : A \vee x : B$

respectively, where A, B are the classes in O_1 and C is the class in O_2. Such a union mapping can be simply decomposed into two subclass mappings as such:

$x : C \leftarrow x : A$

$x : C \leftarrow x : B$

Thus, we only consider the intersection mapping. As we can see, only instances explicitly asserted to be instance of both A and B or for which membership of both classes can be derived are actually mapped to class C. Different ways of relating classes in one ontology are asserting a subclass relationship and asserting class equivalence.

Now, consider the ontologies O_1 and O_2. O_1 consists of the classes $Animal$ and $LegalAgent$; O_2 consists of the class $Human$. Typically, one can relate the classes in the following way:

$x : Human \leftarrow x : Animal \wedge x : LegalAgent$

However, again we need to know for a specific individual that it is both an $Animal$ and a $LegalAgent$. However, this information might not follow from the ontology.

Now, consider:

$\forall x, y \exists z.z : Human \leftarrow x : Animal \wedge y : LegalAgent$

For each combination of an animal and a legal agent, a new human is created during inference, because of the existential. Let's rewrite the formula using a function symbol:

$f(x, y) : Human \leftarrow x : Animal \wedge y : LegalAgent$

This formula has exactly the same result.

Notice that this condition can be seen as a join in database terms. A join typically has conditions on which to join. Say we have a condition that if the name of the animal and the name of the legal agent coincide, then we can map the individual to a human:

$f(x, y) : Human \leftarrow x : Animal \wedge y : LegalAgent \wedge x.name = y.name$

Notice that statements like this are beyond the expressive power of Description Logics. A rules such as this can be expressed using the Semantic Web Rule Language SWRL [18], however, it is very cumbersome, because new terms can only be created using existential value restrictions, instead of using either existentials directly for a named class or using function symbols[1].

Notice that a join mapping is naturally required when combining more than two ontologies. If the classes $Animal$ and $LegalAgent$ would come from two completely dif-

[1]Note that this problem is overcome in the new First-Order Logic extension of SWRL: http://www.daml.org/2004/11/fol/

ferent ontologies, a join has to be created to combine the classes and create a new class $Human$.

As an example we demonstrate the difference between a class intersection mapping and a class join mapping. Say, we have two source classes A and B and a target class C. A class intersection is specified as such:

classMapping(and(A B) C)

The interpretation of this mapping is roughly as follows: every individual that is an instance of *both* A and B is consequently also an instance of C. However, this means that the fractions of the classes A and B which correspond with C already have to be specified as instances of both A and B. In a single ontology, assuming the ontology has been modeled perfectly, this is feasible. However, when dealing with multiple ontologies, this cannot be assumed, and even within one ontology, the classes are not necessarily related to each other.

Furthermore, if A and B are actually not related to each other, this mapping would not work. Say A and B correspond with (disjoint) parts of C. In this case, clearly A and B do not relate to each other, only via C. In this case, A and B have to be joined to create new instances for the class C. This can be specified in the following way:

classMapping(join(A B $\{condition\}$) C)

Notice that in order to do a join, a $condition$ on the join has to be given in order to identify which instances of A and B are to be joined. The $condition$ is between curly brackets to indicate that it is a formula in the logical language.

2.1.2 Attribute - class mapping

We conjecture that an often occurring mapping pattern is that of relating an attribute with a class. We illustrate the pattern with the following example:

Example 2.1. Say, we have an ontology O_1 with a class $Person$, which has an attribute (similar: universal value restriction in Description Logic) $marriedTo$, which has as its range $Person$.

Say the target ontology O_2 has a class $Human$ with no attributes and a class $Marriage$ with the attributes $hasParticipant$ with cardinality 2 and $hasDate$, which is the date of the marriage.

Clearly, the class $Person$ can be mapped to the class $Human$:

$x : Human \leftarrow x : Person$

However, to relate the attribute $marriedTo$ to class $Marriage$ is harder. We can write the following mapping rule (where the attribute $marriedTo$ is a binary predicate):

$Marriage(f(x, y)) \wedge hasParticipant(f(x, y), x) \wedge hasParticipant(f(x, y), y) \leftarrow Person(x) \wedge marriedTo(x, y)$ $\square$

Notice that the final attribute-class mapping could not have been written using an existential quantifier, because then there is no control over the newly constructed term.

Notice also that in the final rule, we did not explicitly state that x and y must be instances of $Human$, since this naturally follows from the first mapping rule and the range restriction on property $marriedTo$.

2.1.3 Class - instance mapping

Depending on the point-of-view of the ontology engineer, an object can be either a class, an instance, an attribute or a relation or perhaps even a constraint, although we do not expect this to be very common and will disregard it in our further treatment.

In the previous section, we have seen an example of an attribute - class mapping. In this section, we will show a class - instance mapping, which we expect to also be common on the Semantic Web [31].

$x :: Airplane \leftarrow x : AirplaneType$

Thus, this rule states that each instance of the concept $AirplaneType$ is actually a subclass of the concept $Airplane$. We can expect this kind of modeling to be common on the Semantic Web, because for some task, the user might want to query all airplane types manufactured by a certain manufacturer, which for a different task, one would want to query for all airplanes of a specific type in service with a specific airline. This kind of modeling might already be used inside one ontology, but can certainly be expected when different ontologies are mapped.

2.1.4 Mapping based on conditions of the target ontology

One might want to express in a mapping that instances can only be translated to the target ontology if certain conditions hold with respect to the target ontology. Alternatively, one might only want to transform a certain instance if the particular instance does not already occur in the target ontology.

One such example (with *not* being default negation) is:

$x : Human \leftarrow x : Person \wedge not\ (y : Human \wedge x.name = y.name)$

This can be rewritten as such:

$x : Human \leftarrow x : Person \wedge not\ y : Human$

$x : Human \leftarrow x : Person \wedge not\ x.name = y.name$

2.1.5 Mapping with built-in(aggregate)s

[21] shows that mappings with aggregate functions can be expected to occur. In their example, they relate and attribute `spouseIn` with an attribute `noMarriages`. Essentially, the number of values for the attribute `spouseIn` is counted to determine the value for the attribute `noMarriages`. We can illustrate this with the following rule (with aggregate function `aggr:count`):

$noMarriages(x, z) \leftarrow Individual(x) \wedge spouseIn(x, y) \wedge aggr : count(z, y)$

The aggregate function is used to count the number of values for the $spouseIn$ attribute to determine the number of marriages that the individual is involved in.

Besides aggregate function, we expect many more built-in functions be required to manipulate data values. For example concatenating strings (first- and lastName vs fullName) or basic arithmetic.

2.1.6 Introducing Terms in the Translation

When translating sets of facts (instances), it is often necessary to introduce new terms. Introducing new terms can be done using existential quantifiers, function symbols (term constructors) or special term generating functions (typically implemented with a built-in predicate).

Mapping with existentials We demonstrate mapping with existentials using an example taken from [13].

$\forall a, t1.@yale_bib : Inproceedings(a) \wedge String(t1) \wedge (booktitle(a, t1) \leftrightarrow (\exists p.Proceedings(p) \wedge contains(p, a) \wedge @cmu_bib : inProceedings(a, p) \wedge @cmu_bib : booktitle(p, t1)))$

This formula is a so-called bridge axiom. This particular axiom forms a bridge between the property `Inproceedings` in ontology `yale_bib` and the property `inProceedings` in ontology `cmu_bib`. In `yale_bib`, the property refers to a string containing the title of the proceedings, whereas in `cmu_bib` the property refers to another individual, which is actually an instance of the class `Proceedings`. Since there is no individual in the former ontology corresponding to the actual proceedings, a new individual is created during inference because of the existentially quantified variable `p`.

Mapping with term generating functions [13] also shows how the above example can be written down with term generating functions:

$\forall a, t1.@yale_bib : Inproceedings(a) \wedge String(t1) \wedge (booktitle(a, t1) \leftrightarrow (contains(@control : aProc(a), a) \wedge Proceedings(@control : aProc(a)) \wedge @cmu_bib :$

$inProceedings(a, @control : aProc(a)) \wedge @cmu_bib : booktitle(@control : aProc(a), t1)))$

In the example, `control:aProc` is a built-in function, which generated a new term on the basis of the terms in the input of the function.

Mapping with function symbols The above example can be written down with function symbols as term constructors:

$\forall a, t1.@yale_bib : Inproceedings(a) \wedge String(t1) \wedge (booktitle(a, t1) \leftrightarrow (contains(f(a), a) \wedge Proceedings(f(a)) \wedge @cmu_bib : inProceedings(a, f(a)) \wedge @cmu_bib : booktitle(f(a), t1)))$

2.1.7 Dummy Mappings

Dummy mappings are coming when we considers mapping between versions of a same ontology. In such a scenario, the mapping is created from the change log of the changed ontology. The initial mapping from an empty change log is obviously made of equivalent mappings between each entities of the two identical versions. As changes are made, the mapping specification is changing. When an entity is dropped from the ontology, the mapping of this entity is then pointing to nothing. On the other way around a concept appearing in the new version is initially related to a Null concept in the old ontology. This is what we call a dummy mapping. Let's have a look to an example, the initial ontology contains the concept $human$. The initial mapping is the following:
$O_{v1} : human \leftrightarrow O_{v2} : human$ Suppose now that the concept human is dropped of the second version of the ontology, the mapping must reflect this change, it now points to a dummy class.
$O_{v1} : human \leftrightarrow O_{v2} : NULL$

Chapter 3

The Mapping Language

In this chapter we present the mapping language in the form of the abstract syntax. Appendix C presents a first-order reference semantics for the mapping language through a mapping to first-order logic. This semantics is given to enhance the understanding of the mapping language and to provide an intuition as to the intended meaning of the constructs. We do not, however, require all users of this mapping language to follow this reference semantics, because of the differences in semantics between ontology languages.

In this deliverable presents the last improvements made to the language since the last version D4.3.1. The mapping language syntax has been slightly modified to better express what the constructs means. See for example the directionality od the mapping rule. Apart from the base language to express alignments we add fields to cope with mapping generated automatically (see the "measure") and to specify the transformation of the instances happening when the mapping is used at run-time. This modifications and improvements are taking into account the state of the art in ontology mapping specification and improve it. Finally we give a rdf vocabulary which can be used as a syntax for the language, giving it support from the rdf management tools.

3.1 Base Language

The abstract syntax is written in the form of EBNF, similar to the OWL Abstract Syntax [25]. Any element between square brackets '[' and ']' is optional. Any element between curly brackets '{' and '}' can have multiple occurrences.

Each element of an ontology on the Semantic Web, whether it is a class, attribute, instance, or relation, is identified using a URI [3]. In the abstract syntax, a URI is denoted with the name **URIReference**. We define the following identifiers:

mappingID ::= **URIReference**

ontologyID ::= **URIReference**
classID ::= **URIReference**
propertyID ::= **URIReference**
attributeID ::= **URIReference**
relationID ::= **URIReference**
individualID ::= **URIReference**

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

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

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

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

mappingdocument ::= 'MappingDocument(' [**mappingID**]
{ 'source(' **ontologyID** ')' }
'target(' **ontologyID** ')'
{ **directive** } ')'

A mapping consists of a number of annotations, corresponding to non-functional properties in WSMO [28], and a number of mapping expressions. The creator of the mapping is advised to include a version identifier in the non-functional properties.

directive ::= **annotation**
|**expression**

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

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

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

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

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

expression ::= ’classMapping(’ ’unidirectional’|’bidirectional’ { **annotation** }
classExpr **classExpr** { **classAttributeMapping** }
{ **classCondition** } [’{’ **logicalExpression** ’}’] ’)’

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

classAttributeMapping ::= ’attributeMapping(’ ’unidirectional’|’bidirectional’ **attributeExpr**
attributeExpr { **attributeCondition** } ’)’

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

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

expression ::= ’instanceMapping(’ **individualID** **individualID** ’)’

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

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

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

expression ::= '{' **logicalExpression** '}'

For class expressions we allow basic boolean algebra. This corresponds loosely with Wiederhold's ontology algebra [36]. Wiederhold included the basic intersection and union, which correspond with our and and or operators. Wiederhold's difference operator corresponds with a conjunction of two class expressions, where one is negated, i.e. for two class expressions C and D, the different $C - D$ corresponds with and(C,not(D)).

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

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

Attribute expressions are defined as such, allowing for inverse, transitive close, symmetric closure and reflexive closure, where inverse(A) stands for the inverse of A, symmetric(A) stands for the symmetric closure of A[1], reflexive(A) stands for the reflexive closure of A[2] and trans(A) stands for the transitive closure of A:

attributeExpr ::= **attributeID**
|'and(' **attributeExpr** **attributeExpr** { **attributeExpr** } ')'
|'or(' **attributeExpr** **attributeExpr** { **attributeExpr** } ')'
|'not(' **attributeExpr** ')'
|'inverse(' **attributeExpr** ')'
|'symmetric(' **attributeExpr** ')'

[1]Notice that the symmetric closure of an attribute is equivalent to the union of the attribute and its inverse: or(A inverse(A)).

[2]The reflexive closure of an attribute A includes for each value v in the domain a tuple with equivalent domain and range v: $\langle v, v \rangle$.

|'reflexive(' **attributeExpr** ')'
|'trans(' **attributeExpr** ')'

Relation expressions are defined similar to class expressions. The arity of a relation migh be indicated.

relationExpr ::= **relationID**{arity}
|'and(' **relationExpr relationExpr** { **relationExpr** } ')'
|'or(' **relationExpr relationExpr** { **relationExpr** } ')'
|'not(' **relationExpr** ')'

classCondition ::= 'attributeValueCondition(' **attributeID** (**individualID** | **dataLiteral**) ')'

classCondition ::= 'attributeTypeCondition(' **attributeID classExpr** ')'

classCondition ::= 'attributeOccurrenceCondition(' **attributeID** ')'

attributeCondition ::= 'valueCondition(' (**individualID** | **dataLiteral**) ')'

attributeCondition ::= 'typeCondition(' **classExpression**) ')'

Especially when mapping several source ontologies into one target ontology, different classes and relations need to be joined. Although apparently similar, a join mapping is fundamentally different from an intersection mapping.

3.2 Extensions

The Alignment format is proposed in [Euzenat2004] and accessible at http://co4.inrialpes.fr/align/format.html. It is designed to express mappings resulting from matching algorithms. To comply with this format we introduce a special annotation. The measure gives an indication about the level of confidence of a mapping rule, it is given as a real number in [0,1]. This field is mainly used when the mappings are the results of a matching algorithm. It may also be proposed to a user in a graphical tool, if the mapping is complex and the user is not certain about his current modeling. The alignment format contains other fields compatible with the mapping language format. The relation gives the kind of relation standing between the two mapped expression. This information is given in the mapping language by the directionality field.

measure ::= 'measure('**value**')'

There is a need to specify transformations of instances in the mappings definition. We propose two way of specifying the transformations. In a first approach, we will propose an extensible library of functions to be included in the pattern library for storage and retrieval. These functions should cover most of the cases presented in the hierarchy of transformation functions[30].

We distinguish two types of transformation functions, the structural transformation is linked to the difference of granularity in the two ontologies: a source ontology may have a concept 'Name' which has two subconcepts 'FirstName' and 'LastName', while in the target ontology only the concept 'Name' exists. Supposing that these concepts all have an attribute 'hasString', the first ontology instances of the concepts 'FirstName' and 'LastName' need to be merged in one instance of the 'Name' concept in the target ontology. To realize the merge operation, the strings corresponding to the attributes values need to be concatenated. When defining the mapping, not only it has to be specified that the concepts are equivalents, but also how the instances will be tranformed. In the last examples two strings have to be merged. 'FirstName' and 'LastName' may become 'FirstName LastName' or 'FirstName.LastName' or even 'F.LastName', 'F' being the initial letter of the 'FirstName'. The mapping specification should then introduce a set of string operators to help specifying how to concatenate, and conversly to split instances attributes.

Another requirement comes from the different encodings of the values in the source and target ontologies. A real in the source ontology may be represented as an integer in the target ontology or the other way around. Different strategies of conversion may be used and must be specified in the mapping: the real numbers may be truncated, rounded up or down in order to be transformed into integers. Another special type is the date format, were durations may be expressed as start and end dates or as number of days for example. A function is then needed to convert from one format to the other.

The next requirement is related to the ontology structure. An ontology O_s might have a concept 'Parent' with a property 'hasChild', whereas the ontology O_t might also have a class 'Parent', but in this case only with the property 'nrOfChildren'. An aggregate function is required to count the number of children in O_s in order to come with a suitable property filler for 'nrOfChildren'. In the same category we add functions to check the values or the presence of others attributes, instances or relations and functions to compare different attributes values.

On the other side of the classification tree we find the transformations based on the values of the instances. In this category we include conversions between different currencies and units, and the operations that may have to be done to convert one value into another. The mapping specification must include a set of common mathematical operators expressive enough to correctly transform the values.

The next figure presents a taxonomy of transformation functions the mapping specification must deal with.

The only aspect not covered by this hierarchy is the dynamicity some transformations require. We cope with this aspect in the second way we propose to handle instance transformation, namely by specifying a web service. Here is the specification of such a function in the mapping language.

transformationFunction ::= ’transformation(’ **functionID** {**param**}’)’
| ’transformation(’ **service IRI** {**param**}’)’

3.3 RDF syntax

At the base of the semantic web stack are found URIs, they are unique identifiers for objects - documents, pieces of informations, real-world objects - allowing unambiguous references. The rdf language use URIs to define a controlled vocabulary for specific purposes. We propose in this section to define a rdf vocabulary for the Sekt ontology mapping language. Using rdf to represent mapping consttructs gives us an acces to the suite of tools developed to manage rdf data: repository, query language, search engine, etc.

Here is the list of keywords. We use ’map’ as a prefix. The URI corresponding to this prefix is the URI of the current document: http://www.omwg.org/TR/d7/d7.2.

Keyword	**Comment**
map#mappingDocument	used to define a mapping document
map#classMapping	mapping between two class expressions
map#attributeMapping	Mapping between two attribute expressions
map#relationMapping	mapping between two relation expressions
map#individualMapping	mapping between two instances
map#classAttributeMapping	mapping between a class expression and an attribute expression
map#classRelationMapping	mapping between a class expression and a relation expression
map#classInstanceMapping	mapping between a class expression and an instance
map#sourceOntology	source ontology
map#targetOntology	target ontology
map#directionality	indicates the directionality of a mapping rule
map#measure	indicates the confidence given to a mapping rule
map#operator	indicates the operator in a complex class, attribute or relation expression
map#condition	specify a condition
map#hasSource	source expression of a mapping rule
map#hasTarget	target expression of a mapping rule
map#hasExpression	source or target expression of a mapping rule. Can also be a sub-expression of an expression
map#logicalExpression	logical expression, represented as a string

The remainder of this section present the translation from the abstract syntax to the rdf syntax. We use the existing rdf constructs to limit the number of introduced keywords. In particular, the documentid is mapped to dc#title, the annotation is mapped to rdf#description.

A,B,C represent identifiers, DV_i stands for an integer value, DV_d stands for a decimal, and DV_s stands for a string data value, and n is an integer number.

Abstract Syntax	**RDF Triples**	**Comments**

T(
MappingDocument(
A
source_exp
target_exp
annotation1 ... annotationn
expression**)**
)
	A rdf#type map#mappingDocument
	T(source_exp,A)
	T(target_exp,A)
	T(annotation$_1$,A) ... T(annotation$_n$,A)
	T(expression
	, A)
		A is the IRI represesenting
		the mapping document given as an id.
		The source and target expressions are ontologies

T(source(B) , A)	A map#sourceOntology B	
T(target(B) , A)	A map#targetOntology B	
T(annotation(C D) , A)	A B T(propertyValue)	annotations are used to indicate informations about the document and the rule, the propertyid B is a dublin core property and then may be used here as an RDF predicate.

T(
classMapping(
annotation$_1$ ... annotatio$_n$
directionality
classExpr
classExpr
classCondition$_1$ ... classCondition$_n$
logicalExpression)
, A)
	A map#classMapping _:X
	T(annotation$_1$, _:X) ...
	T(annotation$_n$, _:X)
	T(directionality, _:X)
	_:X map#hasSource _:Y
	_:X map#hasTarget _:Z
	T(classExpr, _:Y) T(classExpr, _:Z)
	T(classCondition$_1$, _:X) ...
	T(classCondition$_n$, _:X)
	T(logicalExpression, _:X)
		The blank identifier _:X
		denotes a helper node
		to bind the mapping rule
		to the mapping document

T(
attributeMapping(
$annotation_1 ... annotation_n$
directionality
attributeExpr attributeExpr
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpression)
, A)
	A map#attributeMapping _:X
	T(annotation$_1$, _:X) ...
	T($annotation_n$, _:X)
	T(directionality, _:X)
	_:X map#hasSource _:Y
	_:X map#hasTarget _:Z
	T(attributeExpr, _:Y)
	T(attributeExpr, _:Z)
	T(attributeCondition$_1$, _:X) ...
	T(attributeCondition$_n$, _:X)
	T(logicalExpression, _:X)
		The blank identifier_:X
		denotes a helper node
		to bind the mapping rule
		to the mapping document

T(
relationMapping(
annotation$_1$ ... annotatio$_n$
directionality
relationExpr
relationExpr
relationCondition$_1$ ... relationCondition$_n$
logicalExpression)
, A)
	A map#relationMapping _:X
	T(annotation$_1$, _:X) ...
	T(annotation$_n$, _:X)
	T(directionality, _:X)
	_:X map#hasSource _:Y
	_:X map#hasTarget _:Z
	T(relationExpr, _:Y) T(relationExpr, _:Z)
	T(relationCondition$_1$, _:X) ...
	T(relationCondition$_n$, _:X)
	T(logicalExpression, _:X)
		The blank identifier _:X
		denotes a helper node
		to bind the mapping rule
		to the mapping document

T(**instanceMapping(** annotation$_1$ ... annotation$_n$ instance$_1$ instance$_2$, A)	A map#individualMapping _:X T(annotation$_1$, _:X) ... T(annotation$_n$, _:X) _:X map:hasSource instance$_1$ _:X map:hasTarget instance$_2$	The blank identifier _:X denotes a helper node to bind the mapping rule to the mapping document
T(logicalExpression , A)	A map#logicalExpression DV_s8sd:string	

T(classid , A)	A map#hasExpression classid	
T(**and(** $classExpr_1$ ... $classExpr_n$, A)	A map#operator map#and A map#hasExpression _:X_1 T($classExpr_1$, _:X_1) ... A map#hasExpression _:X_n T($classExpr_n$, _:X_n)	Identical for the ’or’ and ’join’ operators, just change to map#or and map#join
T(attributeid , A)	A map#hasExpression attributeid	
T(**and(** $attributeExpr_1$ ... $attributeExpr_n$, A)	A map#operator map#and A map#hasExpression _:X_1 T($attributeExpr_1$, _:X_1) ... A map#hasExpression _:X_n T($attributeExpr_n$, _:X_n)	Identical for the other operators, just change to map#_operator where _operator is the corresponding operator. See the attribute operators list.
T(relationid , A)	A map#hasExpression relationid	
T(**and(** $relationExpr_1$ ... $relationExpr_n$, A)	A map#operator map#and A map#hasExpression _:X_1 T($relationExpr_1$, _:X_1) ... A map#hasExpression _:X_n T($relationExpr_n$, _:X_n)	Identical for the other operators, just change to map#_operator where _operator is the corresponding operator. See the relation operators list.

We can now give a small example of a mapping written both in the abstract and the rdf syntaxes. Here is the header of an example mapping document written in the mapping language abstract syntax

```
MappingDocument(
source(<"http://sw.deri.org/~francois/ontologies/o1">)
target(<"http://sw.deri.org/~francois/ontologies/o2">)
annotation(<"dc:creator">
'http://sw.deri.org/~francois/foaf.rdf')
...
```

Here is the header of a mapping document written in the RDF triples syntax of the mapping language.

```
_"http://sw.deri.org/~francois/mappings/creature2livingThing"
rdf#type
map#mappingDocument

_"http://sw.deri.org/~francois/mappings/creature2livingThing"
  map#sourceOntology
_"http://sw.deri.org/~francois/ontologies/o1"

_"http://sw.deri.org/~francois/mappings/creature2livingThing"
map#targetOntology
_"http://sw.deri.org/~francois/ontologies/o2"

_"http://sw.deri.org/~francois/mappings/creature2livingThing"
dc#creator
_"http://sw.deri.org/~francois/foaf.rdf"
```

A simple example of mapping between two concepts is then given, first in the abstract syntax

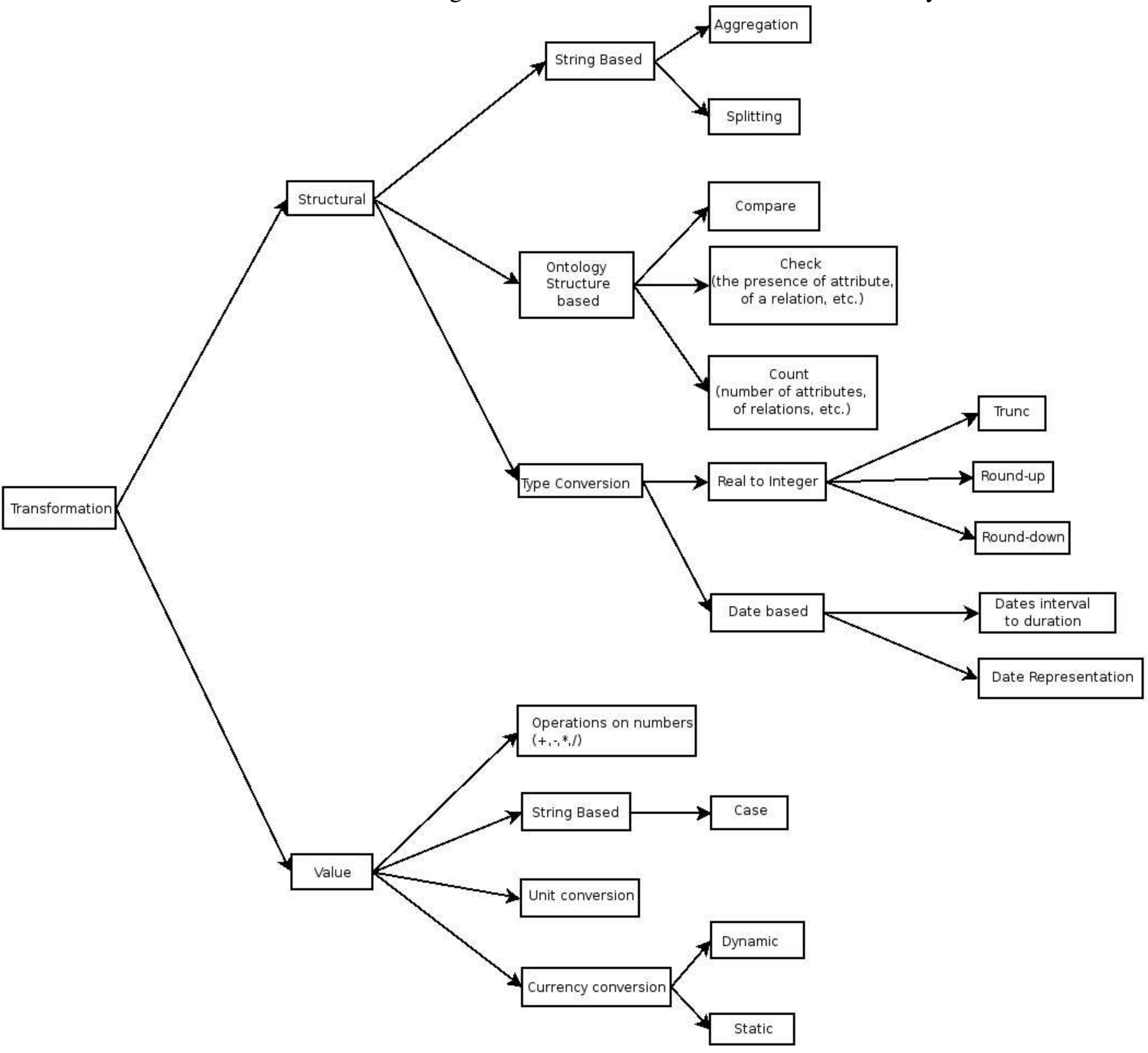
```
classMapping(
  bidirectional
  <"http://ontologies.omwg.org/
           creature#creature">
  <"http://ontologies.omwg.org/
           livingThing#livingThing">)
```

and then in the rdf syntax. We can observe the introduction of an identifier to name the mapping rule.

```
_"http://sw.deri.org/~francois/mappings/creature2livingThing"
map#classMapping
xsd:string^^"rule1"

xsd:string^^"rule1"
map#directionality
xsd:string^^"bidirectional"

xsd:string^^"rule1"
map#hasSource
o1#creature

xsd:string^^"rule1"
map#hasTarget
o2#livingThing
```

The patterns proposed in the next chapter are written using the constructs of this language and using the abstract syntax.

Figure 3.1: Transformation Functions Taxonomy



Chapter 4

Patterns and Patterns Library

4.1 Pattern Template

Patterns are a literary form of software. Their goal is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships let us successfully capture the body of knowledge which defines our understanding of good architectures.

For a short definition of a pattern we quote [1]: *"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution."*

The concept of patterns were first mentioned in architecture by Alexander in 1977 [1], after which it was transferred to computer science. A system should be developed from a human and work perspective. The primary focus is not so much on technology as it is on creating a culture to document and support architecture and design.

Templates are always helpful when you are creating items according to some standard. It makes it easier for others to recognize the form. So it can be re-used if it apply to a similar problem, you can search for new patterns according to a specific scheme.

In the context of ontology mapping, patterns are important to classify the different mappings and to avoid mismatches, as identified in D4.4.1 [8]. In this chapter we develop a template for the description of such ontology mapping patterns.

In the remainder of this chapter we will look into pattern descriptions in software engineering and interaction design. Based on this analysis we develop a template for the description of ontology mapping patterns.

4.1.1 Pattern Templates in Related Work

There are two often-quoted books on the subject of software design using design patterns, both written in the mid nineties of the twentieth century::

Design patterns: Elements of Reusable Object-Orientated Software investigated and described by the so-called Gang of Four in [16] and Coplien reporting on the general use of patterns in software, as well as pattern languages [5].

In 1995 the Gang of Four (GoF) ([16]) has described the following four essential meta elements in a design pattern: **Pattern NAME, PROBLEM Description, SOLUTION and CONSEQUENCES**. The name of a pattern is essential, because it increases the design vocabulary and makes it possible to talk about the pattern. The problem, which is addressed by the patterns, as well as the context in which the problem occurs belongs to the problem description. The abstract description of a design problem and a general arrangement of elements which solves the problem, are given in the solution. Finally, the consequences are the results and trade-off of applying the pattern. Except of Name the three other meta elements refer to the three-part rule which was mentioned at the beginning of this Chapter. These elements contain a complete description of pieces of software and make it easier for a human reader to understand it. Using the scheme it is more effective in retrieving the information that is needed for re-used and shared application.

A year after the GoF published their book on design patterns, Coplien has reported on the general use of patterns in software, as well as pattern languages [5]. He sketched 8 important elements: name, intent, problem, context, forces, solution, sketch, resulting context. The solution is obviously the heart of a pattern, as Alexander cited out already. So Coplien added only name and a shorter problem description in comparison to the GoF. Alias or Known Uses he included in name. For him 'the pattern must work as a seamless piece of literature'. These eight elements are called the minimal set because it includes all necessary information to understand a pattern in software design. In comparison to the GoF template it neglected the community aspects like collaboration, participants, known uses and implementations.

In 2000 Brad Appleton listed 10 essential elements of a pattern in: Patterns and Software: Essential Concepts and Terminology [2]. He focuses clearly on software development and named the object-oriented community. Appleton considers design patters as the basis of software engineering, documenting its best practise and lessons learned. He refer to following 10 elements: Name, problem, context, forces, solution, example, rationale, resulting context, related patterns and known uses. These 10 elements are named the Alexandrian or canonical form because they were first mentioned by Alexander [1].

In 2001 van Welie [34, 33] has described the following pattern template for interaction patterns in user interface design. Based on the minimal elements of Coplien and the template by Gamma, Van Welie added two additional ones for his focus of interaction and usability. These are in detail: Usability Principle to have another term to categorize the patterns according to human-machine interaction and Counterexample as a lack of usage.

In the last year there were some new ideas about patterns published electronically. We will just name them and try to estimate the impact to our work.

Jean-Marc Rosengard and Marian F. Ursu published 'Ontological Representations of Software Patterns' [29] in 2004. This paper analyzes existing pattern representations for automatic organisation, retrieval and explanation of software patterns in the Semantic Web. They suggest nine terms as ontology representations of patterns. In detail they call them: name, also known as, intent, applicability, structure, consequences, implementation, known uses and related pattern. They refer mainly to the pattern template given by Gamma.

We compare the pattern templates described in the previously mentioned literature. Table 4.1 lists all description elements from each of the mentioned approaches and compares the different templates. We try to find matching patches not only by name but also by definition and description and group them according to the meta elements identified by Gamma et al.

As we can see from Table 4.1, the number of elements differs from 13 in object-oriented software design by Gamma et al. to the minimal set of 8 by Coplien as general patterns in software.

Van Welie has special elements dealing with usability criteria see the last elements in column 4 which do not have any equivalents. These are in detail: Usability Principle to have another term to categorize the patterns according to human-machine interaction and Counterexample as a lack of usage.

Gamma et al. focus more on results see bottom of column 2. So Implementation, Collaboration and Participants can be seen as specially important in object-oriented context where modules are often shared and reused.

4.1.2 A Template for Ontology Mapping Patterns

Based on the analysis of patterns templates in the literature we now propose a pattern template for ontology mapping patterns. We structure the elements according to the four meta elements identified by Gamma et al., namely: name, problem, solution and consequences. A brief description of each element shall help the user to document the concept and the work done.

- NAME

  Name A meaningful name to refer to the pattern as a word or single phrase. Nicknames and synonyms can be added under Alias or Also Known as.

template by: Gamma GoF	Coplien	Van Welie	Appleton	belongs to meta element
Pattern Name and Classification	Name	Pattern Name	Name	NAME
also Known as				NAME
Motivation	Problem	Problem Description	Problem	PROBLEM
Applicability	Context	Context	Context	PROBLEM
	Forces	Forces	Forces	PROBLEM
	Solution	Solution	Solution	SOLUTION
Sample Code		Example	Examples	SOLUTION
Intent	Intent	Rationale	Rationale	SOLUTION
Structure	Sketch			
Consequences	Resulting Context		Resulting Context	CONSEQUENCES
Related Patterns		Related Patterns	Related Patterns	CONSEQUENCES
Known Uses		Known uses	Known Uses	CONSEQUENCES
Implementation				
Collaborations				
Participants				
		Usability Principle		
		Counterexample		
13	8	11	10	sum of elements

Table 4.1: Listed pattern elements by different authors

- PROBLEM

  Problem A statement describing the intent, goals and objectives.

  Context The preconditions under which the problem recur, the pattern's applicability.

  Forces A description of relevant forces and constraints, a concrete scenario as motivation.

- SOLUTION

  Solution A description in natural language and mapping language of the pattern.

  Examples Sample application of the pattern.

  Rationale A justifying explanation of steps or rules in the pattern explaining how the forces and constraints are orchestrated.

- CONSEQUENCES

  Resulting Context State or configuration of the system after the pattern has been applied, including the consequences.

  Related Patterns The static and dynamic relationship between this pattern and other within the same pattern language or system.

  Know Uses Describes known occurrences to validate a pattern.

4.2 Patterns

In order to merge ontologies or establish mappings between them, terms in each ontology need to be related to those in the other ontology. Such mappings are necessary for each type of term in an ontology: classes, individuals, relations, and meta-terms. In cases in which the mappings are not one-to-one, either a combination of features in one ontology can be mapped to the meaning of a term in the other ontology or only a unidirectional mapping is possible – with one term defined as being more specific than the other.

A non-exhaustive set of some of the most common types of inter-term mappings for terms in ontologies is presented below.

For the description of the individual mappings, the template described above in Chapter 2 of Name, Problem, Context, Solution, and Examples is followed.

The actual solution description for each pattern consists of two parts, the natural language description of the solution and the abstract syntax for the mapping predicate.

In the mapping syntax specification and the examples, A and B are named classes, C and D are possibly complex class descriptions, R and S are relations, P and Q are attributes, and I and J are individuals. O1 and O2 are namespace qualifiers for the source and target ontologies, respectively. For logical expressions in the examples we use classic first-order logic where a class is represented by a unary predicate, an attribute by a binary predicate and a relation by an n-ary predicate. Furthermore, we allow the usual connectives $\vee, \wedge, \leftarrow, \rightarrow, \leftrightarrow$, the quantifiers $\exists, forall$, the function symbols f, g, h and the variables x, y, z with the usual first-order semantics [15].

4.2.1 Mappings between Classes

This section presents various types of inter-class mappings: equivalence mappings, subclass/superclass mappings, and mappings dependent upon attribute values.

Equivalent Classes

Name: Equivalent Class Mapping
Also Known As: equivalentClassMapping

Problem:
A class in one ontology has the same intention as a class in a second ontology. The terms could have the same name in the different ontologies or different names.

Context:
This is probably the most common pattern in mapping between ontologies.

Solution:
Solution description:
This pattern establishes a bidirectional mapping between classes in two ontologies. Either may be used as the source ontology with the other one being used as the target ontology.

Mapping Syntax:
mapping ::= classMapping(bidirectional A B)

Examples: classMapping(bidirectional O1:Human O2:Person)

Rationale:

Related Patterns: Subclass Mapping, Class Intersection Mapping , Class Union Mapping

Subclass/Superclass Mapping

Name: Subclass Mapping
Also Known As: subClassMapping

Problem:
A class in one ontology is a subclass of a class in a second ontology but there is no functional description of the exact mapping. There is no way of expressing additional properties of the subclass.

Context:
This is a common pattern in which one ontology is more specific than a second ontology. It may also occur when different ontologies specify classes of different intermediate specificities.

Solution:
Solution description:
This pattern establishes a unidirectional mapping from a more specific class in one ontology to a broader class in another ontology. The relation is broadened to allow class expressions in addition to merely class names.

Mapping Syntax:
mapping ::= classMapping(unidirectional A B)

Examples:
classMapping(unidirectional O1:Mammal O2:Vertebrate)
classMapping(unidirectional O2:Vertebrate O1:Chordate)

Rationale:

Related Patterns: Equivalent Class Mapping, Class Intersection Mapping , Class Union Mapping

Class Intersection

Name: Class Intersection Mapping
Also Known As: classIntersectionMapping

Problem:
A class denoted in one ontology is the intersection of two classes in the second ontology.

Context:
This is a common pattern in which one ontology expresses an intersection of classes that may not be useful to distinguish in a second ontology although the individual classes are.

Solution:
Solution description:

This pattern establishes a mapping between a pair of classes in the first ontology and a single class in the other. This pattern is agnostic as to whether the mapping is unidirectional or bidirectional direction of the mapping can be achieved through combination of the pattern with the equivalentClassMapping or subClassMapping pattern.

Mapping Syntax:
mapping ::= classMapping(*direction* and($A_1 \ldots A_n$) B)

Example:
classMapping(bidirectional and(O1:Human O1:FemaleAnimal)
O2:HumanFemale)

Rationale:

Related Patterns: Equivalent Class Mapping, Class Union Mapping

Class Union

Name: Class Union Mapping
Also Known As: classUnionMapping

Problem:
A class denoted in one ontology is the union of two classes in the second ontology.

Context:
This is a common pattern in which one ontology expresses an union of classes that may not be useful to distinguish in a second ontology although the individual classes are.

Solution:
Solution description:
This pattern establishes a mapping between a pair of classes in the first ontology and a single class in the other. This pattern is agnostic as to whether the mapping is unidirectional or bidirectional direction of the mapping can be achieved through combination of the pattern with the equivalentClassMapping or subClassMapping pattern.

Mapping Syntax:
mapping ::= classMapping(*direction* or($C_1 \ldots C_n$) D)

Example:
classMapping(bidirectional
or(O1:PersonBornInCanada O1:PersonWithCanadianParent)
O2:CanadianCitizenByBirth)

Rationale:

Related Patterns: Equivalent Class Mapping, Subclass Mapping, Class Intersection Mapping

Class by Attribute Mapping

Name: Class By Attribute Mapping
Also Known As: classByAttributeMapping

Problem:
A class in one ontology is mapped to a class in the other ontology. However, only those instance which have a particular attribute value are mapped. This pattern is agnostic as to whether the mapping is unidirectional or bidirectional direction of the mapping can be achieved through combination of the pattern with the equivalentClassMapping or subClassMapping pattern.

Context:

Solution:
Solution description:
This pattern establishes a mapping between a class/attribute/attribute value combination in one ontology and a class in another. This pattern is agnostic as to whether the mapping is unidirectional or bidirectional direction of the mapping can be achieved through combination of the pattern with the equivalentClassMapping or subClassMapping pattern.

Mapping Syntax:
mapping ::= classMapping(*direction* A B attributeValueCondition(P o))

Example:
classMapping(bidirectional O1:Human O2:BlueEyedPerson
attributeValueCondition(O1:Vertebrate.eyeColour O1:Blue))

Rationale:

Related Patterns: Equivalent Class Mapping, Subclass Mapping

Class Mapping by Axiom

A subclass mapping can be defined by a more complex rule that specifies which members of the class are included. For example an Uncle is defined as being the brother or brother-in-law of a Parent.

Name: Class Mapping by Axiom
Also Known As: classByAxiomMapping

Problem:
A class in one ontology is mapped to a class in another ontology and the criteria for membership in the class can are specified by an axiom.

Context:
A subclass relationship holds between classes in two ontologies, but the rule defining the subclass cannot be described by any of the above patterns.

Solution:
Solution description:
The two classes in two ontologies are provided along with a statement involving either one class or both classes. This statement is a precondition for the mapping. This pattern is agnostic to whether the mapping is unidirectional or bidirectional. However, if the mapping is bidirectional, the same precondition applies for both directions. The precondition can be an arbitrary logical expression in the language to which the mapping language is grounded. For illustrative purposes, we use first-order logic in the example (X is the meta-variable which represents the instance of the classes in the mapping).

Mapping Syntax:
mapping ::= classMapping(*direction* A B { **logicalExpression** })

Examples:
classMapping
(unidirectional O1:Person O2:Uncle {
$\exists s, k$:
(O1:Person.brother(s X) $\vee$ O1:Person.brotherInLaw(k X)) $\wedge$
O1:Animal.parent(k s) })

Rationale:

Related Patterns: Equivalent Class Mapping, Subclass Mapping

Class Join Mapping

The target instances are created based on the source instances in a database-style join operation.

Name: Class Join Mapping
Also Known As: classJoinMapping

Problem:

A number of classes in one (or more) ontology(ies) are mapping to one class in another ontology. There exists some overlap between the classes in the source ontology. However, this overlap has not been made explicit. It is furthermore clear under which condition the source classes overlap.

Context:

Solution:

Solution description:

First, the source classes are given together with the join condition (there need to be at least two classes). Then, the target class is given. A join mapping is always unidirectional and the join must always be given in the source. Note that in the example we use the ontology identifiers S1,...,Sn to indicate the namespaces of the source ontologies (since it is expected that join mappings will be most common in ontology mappings with multiple source ontologies). T depicts the namespace of the target ontology. In the example $X_1, ..., X_n$ are meta variables which depict the instances of the various source classes. Y depicts the newly constructed instances of the target class.

Mapping Syntax:

mapping ::= classMapping(unidirectional join($A_1 \dots A_n$ { **logicalExpression** }) B)

Examples:

classMapping
(unidirectional join(S1:Person S2:Human { X_1.ssn = X_2.ssn }) T:Person)

Rationale:

Related Patterns: Subclass Mapping

Class Attribute Mapping

A class in one ontology may correspond with an attribute in another..

Name: Class Attribute Mapping
Also Known As: classAttributeMapping

Problem:

A class in one ontology is mapped to an attribute in another ontology.

Context:

Solution:

Solution description:

The class in one ontology and the attribute in the other ontology are provided. Typically the class for the target attribute depends on an attribute of the source class and the range of the target attribute depends on a different attribute of the source class. This pattern is agnostic to whether the mapping is unidirectional or bidirectional.

Mapping Syntax:
mapping ::= classAttributeMapping(*direction* A $B.P$ attributeMapping(Q_1 P) attributeClassMapping(Q_2 B))

Examples:
classAttributeMapping
(O1:Marriage O2:Person.marriedTo
attributeMapping(O1:Marriage.partner1 O2:Person.marriedTo)
attributeClassMapping(O1:Marriage.partner2 O2:Person))

Rationale:

Related Patterns: Equivalent Class Mapping, Subclass Mapping

Class Relation Mapping

A class in one ontology may correspond with a relation in another.

Name: Class Relation Mapping
Also Known As: classRelationMapping

Problem:
A class in one ontology is mapped to a relation in another ontology.

Context:

Solution:
Solution description:
The class in one ontology and the relation in the other ontology are provided. There are no constructs in the mapping language for linking the arguments of the relation. For this, a logical expression need to be used (X is the variable representing the instance of the class; $X_1, ..., X_n$ represent the arguments of the relation). This pattern is agnostic to whether the mapping is unidirectional or bidirectional.

Mapping Syntax:
mapping ::= classRelationMapping(*direction* A R { **logicalExpression** })

Examples:
classRelationMapping
(O1:Marriage O2:Marriage
{ $X_1 = X.partner1 \wedge X_2 = X.partner2 \wedge X_3 = X.dateOfMarriage$ })

Rationale:

Related Patterns: Equivalent Class Mapping, Subclass Mapping, Class Attribute Mapping

Class Instance Mapping

A class in one ontology may correspond with an instance in another.

Name: Class Instance Mapping **Also Known As:** classInstanceMapping
Problem: A class in one ontology is mapped to an instance in another ontology.
Context:
Solution: *Solution description:* The class in one ontology and the instance in the other ontology are provided.
Mapping Syntax: **mapping** ::= classInstanceMapping(A o { **logicalExpression** })
Examples:
Rationale:
Related Patterns:

4.2.2 Mappings between Relations

Equivalent Relation Mapping

Name: Equivalent Relation Mapping **Also Known As:** equivalentRelationMapping
Problem: A relation in one ontology has the same intention as a relation in a second ontology and a mapping between the two ontologies is desired.
Context: This is probably the most common pattern in mapping relations between ontologies. The terms could have the same name in different ontologies or different names.
Solution: *Solution description:* This pattern establishes a bidirectional equivalence mapping between relations in two ontologies. Either may be used as the source ontology with the other one being used as the target ontology.
Mapping Syntax: **mapping** ::= relationMapping(bidirectional R S)

Examples:
relationMapping(bidirectional O1:Human.children O2:Person.parentOf)

Rationale:

Related Patterns: Subrelation Mapping, Inverse Relation Mapping

Subrelation – Superrelation Mapping

Name: Subrelation Mapping
Also Known As: subRelationMapping

Problem:
A relation in one ontology holds between two terms in that ontology only when a more general relation should hold between the mapped terms in the second ontology. However, there is no relation in the second ontology with the same meaning as that in the first.

Context:
One ontology needs to be able to describe certain relations to a greater degree of precision.

Solution:
Solution description:

This pattern establishes a unidirectional mapping between relations in two ontologies. The source ontology is the first ontology specified while the second one specified is the target ontology.

Mapping Syntax:
mapping ::= relationMapping(unidirectional R D)

Example:
relationMapping(unidirectional O1:Human.adores O2:Person.likes)

Rationale:

Related Patterns: Equivalent Relation Mapping

Negated Relation Mapping

Name: Negated Relation Mapping
Also Known As: relationNegationMapping

Problem:
A relation in one ontology holds if and only if a relation in another ontology does not hold for arguments which meet the constraints of the relations.

Context:
This pattern is likely to occur for relations dealing with comparisons. It may only occur when negation can be expressed in at least one of the ontologies. For calculating "only if" either a closed world assumption is needed for the predicate being mapped or some other way of determining the negation of the predicate in (at least) limited cases is needed.

Solution:
Solution description:
The pattern establishes a mapping between a relation in one ontology and the negation of a relation in another ontology. This pattern is agnostic as to whether the mapping is unidirectional (*if*) or bidirectional (*if and only if*). Direction of the mapping can be achieved through combination of the pattern with the equivalentRelationMapping or subRelationMapping pattern.

Mapping Syntax:
mapping ::= relationMapping(bidirectional R not(S))

Examples:
relationMapping(bidirectional
O1:Real.greaterThan not(O2:RealNumber.lessThanOrEqual))

Resulting Context:
This pattern establishes a bidirectional negated mapping between relations in two ontologies. If the relation R1 in the first ontology holds between two arguments, R2 does not hold between the mappings of those arguments in the second, and vice versa. Either ontology may be used as the source ontology with the other one being used as the target ontology. The pattern identifies the incompatibility of relations in different ontologies, allowing the mapping of rules and ground statements involving the relations between the two ontologies if negation is allowed in the mapped forms.

Rationale:

Related Patterns: Equivalent Relation Mapping, Subrelation Mapping

---

Relation Mapping by Axiom

Name: Relation Mapping by Axiom

Also Known As: relationByAxiomMapping

Problem:
A relation in one ontology is mapped to a relation in another ontology and common tuples of the relations are specified by an axiom.

Context:
A relationship holds between relations in two ontologies, but the rule defining the set of tuples in both relations cannot be described by any of the above patterns.

Solution:
Solution description:
The two relations in two ontologies are provided along with a statement involving either one relation or both relations. This statement is a precondition for the mapping. This pattern is agnostic to whether the mapping is unidirectional or bidirectional. However, if the mapping is bidirectional, the same precondition applies for both directions. The precondition can be an arbitrary logical expression in the language to which the mapping language is grounded. For illustrative purposes, we use first-order logic in the example ($X_1, ..., X_n$ are meta-variables which represent the arguments of the source relation in the mapping and $Y_1, ..., Y_n$ are meta-variables which represent the arguments of the target relation in the mapping).

Mapping Syntax:
mapping ::= relationMapping(*direction* R S { **logicalExpression** })

Examples:
relationMapping
(O1:DistanceInMiles O2:DistanceInKM {
$Y_1 = X_1 \wedge Y_2 = X_2 \wedge Y_3 = milesToKM(X_3)$ })

Rationale:

Related Patterns: Equivalent Class Mapping, Subclass Mapping

Attribute Transitive Closure

Name:
Attribute Transitive Closure Mapping
Also Known As: attributeTransitiveClosureMapping

Problem:
An attribute in one ontology is the transitive closure of an attribute in a second ontology.

Context:

One ontology describes an attribute which a second one does not include, although the second can express the attribute as a transitive closure of an attribute which it does possess.

Solution:
Solution description:
This pattern establishes a mapping between an attribute in one ontology and its transitive closure in a second.

Mapping Syntax:
mapping ::= attributeMapping(*direction* P trans(Q))

Examples:
attributeMapping(bidirectional trans(O1:Human.parents)
O2:Person.ancestors)
attributeMapping(bidirectional trans(O1:Animal.parents)
O2:Person.ancestors)

Resulting Context:
The pattern can be used to identify a transitive closure mapping between attributes in different ontologies, allowing the mapping of rules and ground statements involving the attributes between the two ontologies.

Rationale:

Related Patterns: Subrelation Mapping, Equivalent Relation Mapping

---

Inverse Attribute Mapping

Name: Inverse Attribute Mapping
Also Known As: attributeInverseMapping

Problem:
An attribute in the one ontology has the same meaning as an attribute in the second ontology except the domain and range are reversed.

Context:
This is a common pattern in mapping attributes between ontologies.

Solution:
Solution description:
Uses of the attribute in one ontology have their argument order reversed when mapped to the second ontology. Either ontology may be used as the source ontology with the other one being used as the target ontology.

Mapping Syntax:
mapping ::= attributeMapping(*direction* P inverse(Q))

Examples:

attributeMapping(
O2:RealNumber.lessThanOrEqual inverse(O1:Real.greaterThanOrEqual))

Resulting Context:
The pattern can be used to identify an inverse relationship between attributes in different ontologies, allowing the mapping of rules and ground statements involving the attributes between the two ontologies.

Rationale:

Related Patterns: Equivalent Relation Mapping, Subrelation Mapping

Attribute Value Mapping

Attribute values are restricted in some ontological languages to individuals and instances of datatypes, while other languages permit relations, and classes as well. Some languages distinguish attribute values as a special class of individual.

Name: Attribute Value Mapping
Also Known As: attributeValueMapping

Problem:
Character strings and numbers are often used as attribute values in an ontology instead of reifying the individuals, classes, or relations which they represent. Thus, there is often a one-to-one correspondence between an attribute value in two ontology in the context of some attribute. Either attribute value might be a text string, number, individual, or class.

Context:
This is the most common pattern in mapping between attribute values.

Solution:
Solution description:
This pattern establishes a mapping between attribute - attribute value pairs in two ontologies. Either ontology may be used as the source ontology with the other one being used as the target ontology.

Mapping Syntax:
mapping ::= attributeValueMapping(*direction* $A\ I\ B\ J$)

Examples:
attributeValueMapping(bidirectional
(attributeValueCondition O1:PhysObj.color "FF0000")
(attributeValueCondition O2:Object.hasColour "SaturatedRed"))

attributeValueMapping(bidirectional
(attributeValueCondition O1:Address.country Ireland)
(attributeValueCondition O2:Address.country "IE"))

Resulting Context:
The pattern maps an attribute in the source ontology to an attribute value in the target ontology in the context of specified attributes).

Rationale:

Related Patterns:

4.2.3 Mappings between Individuals

Equivalent Individual Mapping

The most common type of mapping that is established between individuals in two ontologies is mapping equivalent terms to each other. The terms could have the same name in different ontologies or different names.

Name: Equivalent Individual Mapping
Also Known As: equivalentIndividualMapping

Problem:
An individual in the one ontology has the same meaning as an individual in the second ontology. The terms could have the same name in different ontologies or different names.

Context:
This is probably the most common pattern in mapping between individuals.

Solution:
Solution description:
This pattern establishes a bidirectional mapping between individuals in two ontologies. Either may be used as the source ontology with the other one being used as the target ontology.

Mapping Syntax:
mapping ::= individualMapping(I J)

Examples: individualMapping(O1:GWBush O2:Dubya)

Resulting Context:

The pattern maps an instance in the source ontology to an instance in the target ontology. This amounts to far more than an equality assertion between the instances in the two ontologies. It entails the mapping of every statement involving the instance in the source ontology into an equivalent statement in the target ontology, if possible, otherwise to an entailed statement (again, if possible).

Rationale:

Related Patterns:

Equivalent Relation Instance Mapping

Name: Equivalent Relation Instance Mapping
Also Known As: equivalentRelationInstanceMapping

Problem:
A tuple of a relation in the one ontology has the same meaning as a tuple of a relation in the second ontology.

Context:
This is probably the most common pattern in mapping between relation tuples.

Solution:
Solution description:
This pattern establishes a bidirectional mapping between tuples of relations in two ontologies. Either may be used as the source ontology with the other one being used as the target ontology.

Mapping Syntax:
mapping ::= relationInstanceMapping($R(I_1, \ldots, I_n)$ $S(J_1, \ldots, J_n)$)

Examples: relationInstanceMapping(O1:distanceInKM(location1, location2, 18) O2:distanceInMiles(location1, location2, 11))

Resulting Context:

Rationale:

Related Patterns:

4.2.4 Attribute Value – Class Equivalence

Name: Attribute Value – Class Mapping

Problem:
Character strings and numbers are often used as attribute values in an ontology instead of reifying the individuals, classes, or relations which they represent. Thus, there is often a one-to-one correspondence between an attribute value in one ontology and a class in another ontology. The attribute value applies to an instance in the first ontology if and only if the mapping of that instance is a member of the class in the second ontology.

Context:
One ontology commonly uses attribute values to make distinctions that another ontology makes using classes.

Solution:
Solution description:
This pattern establishes a bidirectional mapping between an attribute value in one ontology and the attribute with which it is associated and a class in a second ontology. Either may be used as the source ontology with the other one being used as the target ontology.

Abstract Syntax:
'attributeClassMapping(' **attributeCondition classExpr** ')'

Mapping Syntax:
mapping ::= classMapping(bidirectional C D attributeOccurence(A))

Examples:
attributeClassMapping (attributeValueCondition(O1:Person.degreeType "PhD")
O2:PersonWithPhDDegree)

Resulting Context:
The pattern establishes a mapping between the set of all instances with a given attribute value in one ontology with a class in a second ontology.

Rationale:
This is a common type of mapping when two ontologies use different philosophies for use of attributes vs. definition of subclasses.

Related Patterns:

Subattribute / SuperAttribute Value Mapping

Name: Subattribute Value Mapping

Problem:

An attribute value in the one ontology has a more restrictive meaning than an attribute value in the second ontology in the context of some attribute. Either attribute value might be a text string, number, individual or class.

Context:
Anything that has the subattribute value with respect to a given attribute in the first ontology has the superattribute value (with respect to the corresponding attribute) in the second ontology, but not the other way around. This allows a mapping in one direction, but not the other.

Solution:
Solution description:
This pattern establishes a unidirectional mapping between an attribute value condition in one ontology and one in another.

Abstract Syntax:
'subAttributeValueMapping(' **attributeCondition attributeCondition** ')'

Mapping Syntax:
mapping ::= attributeValueMapping(unidirectional $A\ I\ B\ J$)

Example:
subAttributeValueMapping(
(attributeValueCondition O1:PhysObj.color O1:DeepGreen)
(attributeValueCondition O2:Object.hasColour O2:GreenColour))

Resulting Context:
The pattern maps an attribute in the source ontology to an attribute value in the target ontology in the context of specified attributes).

Rationale:
Attribute - Attribute Value pairs are expressed as attributeConditions to modularize this pattern, allowing it to be based on a binary instead of quaternary relation.

Related Patterns: Equivalent Attribute Value Mapping

4.2.5 Dummy Mapping

Name: dummyMapping

Problem:
A class, attribute or relation in one ontology is dropped in the next version of this ontology.

Context:
The dropped entity must be related to a dummy entity in the new version of the ontology

Solution:
Solution description:
We introduce a `null` entity in the new version.

Abstract Syntax:
'ClassMapping(' **ClassId null**
'AttributeMapping(' **AttributeId null**
'RelationMapping(' **RelationId null**
mapping ::= classMapping(bidirectional C $null$)
mapping ::= attributeMapping(bidirectional C $null$)
mapping ::= relationMapping(bidirectional C $null$)

Examples:
dummyMapping(
(Ov1:Human Ov2:null))

Resulting Context:
The pattern can be used to tell the user that the dropped concept has no representation in the new version of the ontology having for a consequence that the possible instances of this concept are no longer related to any concept.

Rationale:
Three null entities are introduced: one for concepts, one for attributes and one for relations in the ontology new version.

Related Patterns:

4.3 A hierarchical organisation of the Patterns Library

In this section we present a hierarchical organisation of the elementary mapping patterns described in this deliverable.

The hierarchy has the form of a classification hierarchy, rather than a formal taxonomy. This means that the links in the hierarchy do not have a formal meaning. This means that a pattern lower in the hierarchy is either a specialisation or a part of the pattern higher in the hierarchy. The hierarchy of elementary mapping patterns is presented in Table 4.24.

These mapping patterns have a correspondence in the mapping language. However, there is not a one-to-one correspondence between mapping patterns and keywords in the mapping language. We decided to keep the mapping language itself concise and to allow only a limited number of keywords. By combining these keywords, the mapping patterns themselves can be written down in the mapping language, see Table 4.3. Notice that not all patterns have a corresponding statement in the mapping language. This is because several patterns (e.g. classMapping, relationMapping) have been introduced mostly to

- classMapping
    - equivalentClassMapping
    - subClassMapping
    - classIntersectionMapping
        - equivalentClassIntersectionMapping
        - subClassIntersectionMapping
        - ...
    - classUnionMapping
        - equivalentClassUnionMapping
        - ...
    - classByAttributeMapping
    - classByAxiomMapping
    - classJoinMapping
    - classAttributeMapping
    - classRelationMapping
    - classIndividualMapping
- relationMapping
    - subRelationMapping
    - equivalentRelationMapping
    - attributeMapping
        - attributeTransitiveClosureMapping
        - attributeInverseMapping
        - attributeValueMapping
            - equivalentAttributeValueMapping
            - subAttributeValueMapping
    - relationNegationMapping
        - subRelationNegationMapping
        - ...
    - relationByAxiomMapping
- individualMapping
    - equivalentIndividualMapping
    - equivalentRelationInstanceMapping
- dummyMapping

Table 4.24: Hierarchical Organization of Mapping patterns

Mapping Pattern	**Corresponding Mapping Statement**
equivalentClassMapping	classMapping(two-way A B)
subClassMapping	classMapping(A B)
equivalentClassIntersectionMapping	classMapping(two-way and($A_1 \ldots A_n$) B)
equivalentClassUnionMapping	classMapping(two-way or($A_1 \ldots A_n$) B)
subClassIntersectionMapping	classMapping(and($A_1 \ldots A_n$) B)
subClassUnionMapping	classMapping(or($A_1 \ldots A_n$) B)
subClassByAttributeMapping	classMapping(A B attributeOccurence(P))
subClassByAxiomMapping	classMapping(A B { $axiom$ })
subRelationMapping	relationMapping(R B)
equivalentRelationMapping	relationMapping(two-way R B)
attributeTransitiveClosureMapping	attributeMapping(two-way P trans(Q))
attributeInverseMapping	attributeMapping(two-way P inverse(Q))
equivalentAttributeValueMapping	attributeValueMapping(two-way P I Q J)
subAttributeValueMapping	attributeValueMapping(P I Q J)
subRelationNegationMapping	relationMapping(R not(S))
subRelationByAxiomMapping	relationMapping(R S { $axiom$ })
equivalentIndividualMapping	individualMapping(I J)
equivalentRelationInstanceMapping	individualMapping($R(I_1, \ldots, I_n)$ $S(J_1, \ldots, J_n)$)

dummyMapping	ClassMapping(bidirectional A null)
	AttributeMapping(bidirectional A null)
	relationMapping(bidirectional A null)

Table 4.25: Correspondence between mapping patterns and statements in the mapping language

structure the patterns[1]. Furthermore, these patterns do not have a clear meaning. For example, when using the pattern classMapping, it is not immediately clear whether class equivalence or class subsumption is intended. Therefore, the additional patterns subClassMapping and equivalentClassMapping have been introduced to clarify which kind of mapping is intended.

In the Table 4.3, A_i, B_i are class names, R_i, S_i are relation names and I_i, J_i are individual names. Furthermore, P, Q are attribute names, where attributes are a special kind of relations, namely, binary relations with a defined domain.

[1]Notice that these "abstract" patterns can also be used to structure the process of specifying mappings. When the developer (or matching algorithm) identifies two classes to be similar, an abstract classMapping can be designated. This can be refined at a further stage. In the specific scenario of ontology matching, it can be envisioned that a matching algorithm would discover such abstract mappings and that the mapping engineer would specify the mapping more precisely.

Chapter 5

Implementation

5.1 Introduction

In this section we presents the tools that have been implemented to cope with the mapping patterns library as well as with the ontology mapping language presented in this deliverable. For a detailed description of the other tools for the mediation in SEKt, we refer the reader to the deliverable D4.5.3. The Ontology Mapping Store give the pattern library a physical support to store the patterns, providing advanced search fuctionalities. The mapping language API give support to the mapping language, providing a parser to parse the constructs of the language, an object model to manipulte the mapping documents using Java constructs, a module allowing to export the mapping documents in different ontology languages, as well as an adapter interface giving the possibility to the mapping API to deal with different ontology language manipulation programs.

The tools presented in this section are available at the following URI: http://www.sekt-project.org/Members/francois.scharffe/tools/.

5.2 Mappings and Mapping Patterns Store

The main objective of the mapping store is to allow retrieval and reuse of existing mappings. This is particularly useful in Semantic Web context, when one can expect large number of partial alternative mappings between a couple of ontologies. The user trying to setup his mapping should be able to locate and compare concrete mappings between ontology elements in interest to her particular case. The mapping store was designed as a high-performance open-source back end tool, and is implemented in Java. The storage is based on Apache Lucene IR engine, which is also open-source. The open-source approach was chosen because of its advantages for easy adoption by the community. The store supports the Mapping Language and is accessible from the Mapping Tool as well as via Java API as a separate service component. Thus, the mapping store can be used to share mappings between several mapping tools as a common repository.

5.3 Mapping Language API

This section presents the Java API developed and implemented to support the mapping language. We present the different component constituting the API, namely the parser, the object model, the export module and the adapter interface. Figure 5.1 presents the mapping API and its interactions.

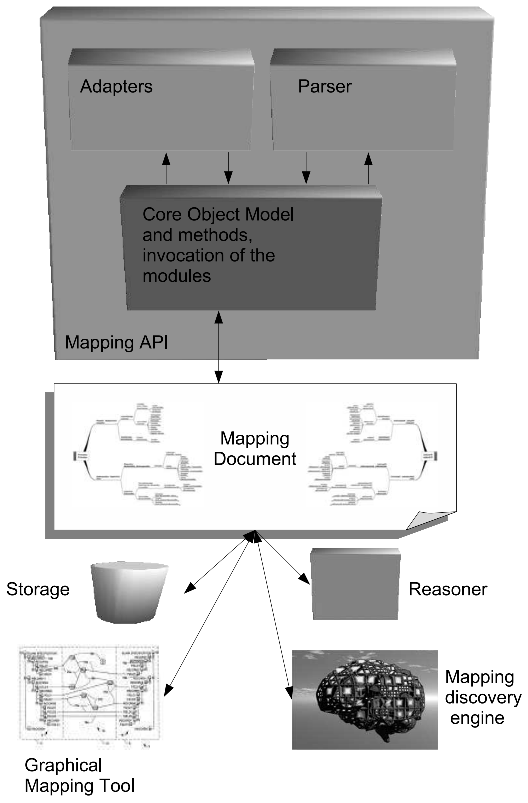


Figure 5.1: Mapping API in situ

5.3.1 Parser

The mapping language is written Lisp-like syntax. This syntax presents the advantage to be easily readable by a human, it is however more complicated to make it read by a computer as we cannot get support from the numbers of tools developped for XML like syntaxes. We then have to implement our own parser. We used for this Sablecc[1]. Sablecc

[1]http://www.sablecc.org

is a compiler compiler. From a given grammar specification in Extended Backaus-Naurus Form (EBNF) it generates a parser. We give the grammar specification in EBNF of the mapping language in Annex D.

5.3.2 Object Model

Once the parser generated, it is of no use as a stand alone tool, the language parsed must instanciate an object model so that the mapping document may have a representation in memory that maig be manipulated. We had in that perspective to design such a model of classes an properties following the structure of the language. We have then a top class `MappingDocument` which will contain the information about the document like the source ontology, the target ontology, the different mapping rules and annotations. A tree walker goes around the tree generated by the parser and instaciate the object model with the actual mapping document. Different methods give the possibility to manipulate the mapping, it is possible to modify the annotations, the id of the document, the source and target ontologies, it is possible to add or suppress a rule etc. Once the mapping document edited, it might be exported via the export module.

5.3.3 Export Module

The export module give the possibility to export mapping documents in various gounding formats. It currently allows to export in the abstract syntax of the mapping language, in the ontology web laguage (OWL-DL) and in WSML the Web Services Modelling Language. This implementation gives the possibility to use the mappings at run-time by loading them into a reasoner together with the two mapped ontologies.

5.3.4 Adapters interface

The mapping language and patterns are expected to be root of ontology mapping tools, like it is the case for the Sekt ontology mapping tool Ontomap. Like the different tools are using either diffenret ontology maodels or have different representations of the ontology constructs, we provide an adapter interface which can be implemented to

5.4 Conclusions

We have in this chapter presented the different tools implemented to support the mapping patterns library and its associated mapping language.,It is however good to piont the reader to the electronic documentation of the different programs as well as the program itself.

Chapter 6

Conclusions

In this deliverable we have presented an update to the mapping language which was presented in the previous version of this deliverable (D4.3.1). We have also included the OWL grounding and the WSML grounding as appendices to this deliverable; these were originally presented in other deliverables.

The major updates to the mapping language are the RDF syntax and the extensions with measures and transformation functions. The measures are required to capture the output of certain alignment algorithms, where it is necessary to specify the confidence one has in the mapping. The transformation functions are used to capture certain data transformations which are necessary when transforming data from one representation to the other, e.g., transforming temperature measures represented using the Fahrenheit scale to temperature measures represented using the Celcius scale.

One of the motivations for introducing the confidence measure was alignment with the Knowledge Web API for ontology alignment [14]. The alignment API and the mapping language presented in this deliverable are currently the leading representation format for representing correspondences between ontologies on the Semantic Web. Where our mapping language focuses on representing possibly complex correspondences between ontologies, the alignment API was designed with the goal of representing the output of alignment algorithms in mind. Therefore, the alignment API is especially suited for representing simple mappings, as well as confidence measures for these mappings, whereas out mapping language is suitable for representing more complex mappings. It is an ongoing effort to integrate the Knowledge Web alignment API with the SEKT mapping language in order to come up with one uniform representation mechanism for ontology mappings.

Besides the mapping language, we have presented the mapping language API, which provides basic parsing and serialization support for the mapping language and which currently implements the OWL DL and WSML-Rule groundings. An extension is foreseen to extend the OWL DL grounding to SWRL, in order to enable reasoning over expressive mappings using the KAON2 reasoner, which is the reasoner used in SEKT.

Bibliography

[1] Christopher Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*, volume 2 of *Center for Environmental Structure Series*. Oxford University Press, New York, New York, USA, 1977.

[2] B. Appleton. Patterns and software: Essential concepts and terminology. http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html, 2000.

[3] T. Berners-Lee, R. Fielding, U. C. Irvine, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396, Internet Engineering Task Force, 1998.

[4] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 600–607, Madison, Wisconsin, USA, 1998. MIT Press.

[5] James O. Coplien. *Software Patterns*. SIGS Books, New York, New York, 1996.

[6] Jos de Bruijn, Dieter Fensel, Uwe Keller, Michael Kifer Holger Lausen, Reto Krummenacher, Axel Polleres, and Livia Predoiu. Web service modeling language (WSML). W3C Member Submission 3 June 2005, 2005.

[7] Jos de Bruijn, Francisco Martín-Recuerda, Dimitar Manov, and Marc Ehrig. State-of-the-art survey on ontology merging and aligning v1. Deliverable D4.2.1, SEKT, 2004.

[8] Jos de Bruijn, Francisco Martín-Recuerda, Axel Polleres, Livia Predoiu, and Marc Ehrig. Ontology mediation management v1. Deliverable D4.4.1, SEKT, 2004.

[9] Jos de Bruijn, Axel Polleres, Rubén Lara, and Dieter Fensel. OWL DL vs. OWL Flight: Conceptual modeling and reasoning on the semantic web. Technical report, Chiba, Japan, 2005.

[10] Mike Dean and Guus Schreiber, editors. *OWL Web Ontology Language Reference*. 2004. W3C Recommendation 10 February 2004.

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

[12] AnHai Doan, Jazant Madhaven, Pedro Domingos, and Alon Halevy. Ontology matching: A machine learning approach. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies in Information Systems*, pages 397–416. Springer-Verlag, 2004.

[13] Dejing Dou, Drew McDermott, and Peishen Qi. Ontology translation by ontology merging and automated reasoning. In *Proc. EKAW2002 Workshop on Ontologies for Multi-Agent Systems*, pages 3–18, 2002.

[14] Jérôme Euzenat. An API for ontology alignment. In *3rd International Semantic Web Conference (ISWC2004)*, pages 698–712, 2004.

[15] M. Fitting. *First Order Logic and Automated Theorem Proving (second edition)*. Springer Verlag, 1996.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Pub., 1995.

[17] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. S-match: an algorithm and an implementation of semantic matching. In *Proceedings of ESWS'04*, number 3053 in LNCS, pages 61–75, Heraklion, Greece, 2004. Springer-Verlag.

[18] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. Available from http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, May 2004.

[19] Michael Kifer, Geord Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4):741–843, 1995.

[20] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *Proc. 27th Int. Conf. on Very Large Data Bases (VLDB)*, 2001.

[21] Alexander Maedche, Boris Motik, Nu no Silva, and Raphael Volz. MAFRA - a mapping framework for distributed ontologies. In *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW-2002*, Madrid, Spain, 2002.

[22] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Developing metadata-intensive applications with rondo. *Journal of Web Semantics*, 1(1), December 2003.

[23] Natalya F. Noy and Mark A. Musen. Smart: Automated support for ontology merging and alignment. Technical Report SMI-1999-0813, Stanford Medical Informatics, 1999.

[24] John Y. Park, John H. Gennari, and Mark A. Musen. Mappings for reuse in knowledge-based systems. In *Proceedings of the 11th Workshop on Knowledge Acquisition, Modelling and Management (KAW 98)*, Banff, Canada, 1998.

[25] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL web ontology language semantics and abstract syntax. Recommendation 10 February 2004, W3C, 2004.

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

[27] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, 2001.

[28] Dumitru Roman, Holger Lausen, and Uwe Keller, editors. *Web Service Modeling Ontology (WSMO)*. 2004. WSMO Final Draft D2v1.0. Available from http://www.wsmo.org/2004/d2/v1.0/.

[29] J.-M. Rosengard and M. F. Ursu. Ontological representations of software patterns. *Lecture Notes in Computer Science (Proceedings of the of KES'04)*, 3215, 2004.

[30] François Scharffe. Omwg d7.2: Mapping and merging tool design. Technical report, 2005.

[31] Guus Schreiber. The web is not well-formed. *IEEE Intelligent Systems*, 17(2), 2002. Contribution to the section Trends and Controversies: Ontologies KISSES in Standardization.

[32] Gerd Stumme and Alexander Maedche. Fca-merge: Bottom-up merging of ontologies. In *7th Intl. Conf. on Artificial Intelligence (IJCAI '01)*, pages 225–230, Seattle, WA, USA, 2001.

[33] Martijn van Welie. *Task-based User Interface Design*. PhD thesis, Vrije Universiteit Amsterdam, 2001.

[34] Martijn van Welie and Gerrit C. van der Veer. Pattern languages in interaction design: Structure and organization. In *Proceedings of Interact '03*, pages 527–534, Zürich, Switserland, 2003. IOS Press.

[35] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. Dublin core metadata for resource discovery. RFC 2413, IETF, 1998.

[36] Gio Wiederhold. An algebra for ontology composition. In *Proceedings of 1994 Monterey Workshop on formal Methods*, pages 56–61, U.S. Naval Postgraduate School, Monterey CA, 1994.

Appendix A

WSML Syntax

In order to allow for reasoning with the mapping language we formally ground the semantics of the mapping language to Logic Programming and, more specifically, to WSML-Rule [6].

We translate the mapping language to a rule language and thus there are several restrictions on the type of rules which can be created. In general, a rule may only have one literal in the head. Rules with a conjunction can easily be rewritten into a number of rules with only a single literal in the head. However, a disjunction in the head cannot be handled. Furthermore, WSML-Rule does not allow for classical negation, only default negation. Default negation may only occur in the body of the rule; not in the head.

All the statements of the mapping language are translated to WSML by the function $t()$ which takes as argument a mapping language statement and returns the corresponding WSML construct. The symbol '$\mapsto$' makes the links between the function call and result returned by this function. Note that the constructs containing '$\mapsto$' are not rules, i.e. '$\mapsto$' does not separate a rule head and a rule body, as it might be suggested by the form of these constructs.

In the mapping, $?x$ and $?y$ are variables. $?x_{new}$ stands for a newly introduced variable. X and Y are meta-variables which are replaced with real variables during the translation. In the class, attribute, and relation expressions 'naf' stands for *negation as failure*.

The **URIReference**s used to identify the **mappingID**, **ontologyID**, **classID**, **propertyID**, **attributeID**, **relationID**, and **individulID** are represented by full URIs in WSML. As a consequence, we have:

$t($**URIReference**$) \mapsto$ _"URIReference"

The **plainLiteral**s have direct correspondents in WSML language as strings (the syntax for the plain literal is the same as for the string in WSML):

$t($**plainLiteral**$) \mapsto$ plainLiteral

Typed literals are transformed to constructed data values, according to the correspondence between XSD data types and WSML datatype wrappers (see Appendix C of [6]):

$t($"*string*"^^*datatype* $) \mapsto$ *datatypeConstructor*(*args*)

The **logicalExpression**s have also a direct correspondent in WSML logical expressions, namely:

$t($**logicalExpression**$) \mapsto$ **expr**

It is possible to use a meta-variable in logical expressions which are nested inside other mapping expression (for example, class mappings). The meta-variable is '?X' and is syntactically substituted in the translation of the mapping language to WSML-Rule:

$t($**logicalExpression**$, X) \mapsto$ **expr**[?X := X]

The annotations in the mapping language correspond to the non-functional properties in WSML; several annotations can be translated in one non-functional properties block:

$t(Annotation_1(P_1\ v_1)...Annotation_n(P_n\ v_n)) \mapsto$
nonFunctionalProperties
    P_1 **hasValue** v_1
    ...
    P_n **hasValue** v_n
endNonFunctionalProperties

A.1 Class mappings

Below, the translations of class mappings are specified. First, a bidirectional class mapping is translated into two unidirectional class mappings. Then, a unidirectional mapping is translated as a rule of subtranslations.

$t($*classMapping*(*bidirectional* classExpr$_1$ classExpr$_2$
    attributeMapping$_1$ ... attributeMapping$_n$
    classCondition$_1$ ... classCondition$_m$
    logicalExpression$_1$ ... logicalExpression$_q$)) $\mapsto$
$t($*classMapping*(*unidirectional* classExpr$_1$ classExpr$_2$
    attributeMapping$_1$ ... attributeMapping$_n$
    classCondition$_1$ ... classCondition$_m$

logicalExpression$_1$ ... logicalExpression$_q$))
$t($*classMapping*(*unidirectional* classExpr$_2$ classExpr$_1$
attributeMapping$_1$ ... attributeMapping$_n$
classCondition$_1$ ... classCondition$_m$
logicalExpression$_1$ ... logicalExpression$_q$))

$t($*classMapping*(*unidirectional* classExpr$_1$ classExpr$_2$
attributeMapping$_1$ ... attributeMapping$_n$
classCondition$_1$ ... classCondition$_m$
logicalExpresion$_1$ ... logicalExpresion$_q$)) $\mapsto$
$t($classExpr$_2, ?x)$ **impliedBy** $t($classExpr$_1, ?x)$ 'and'
$t($attributeMapping$_1, ?x)$ 'and' ... 'and' $t($attributeMapping$_n, ?x)$
$t($classCondition$_1$, $?x)$ 'and' ... 'and' $t($classCondition$_m$, $?x)$
$t($logicalExpression$_1, ?x)$ ... $t($logicalExpression$_q, ?x)$ '.'

In the mapping language, for different class expressions different translations are required. There are no explicit constructs for representing the intersection, union, difference and join operations in WSML. Therefore, we have to create a new concept and to write for it the WSML logical expression that defines the intersection, union, complement, and join, respectively. Note that the or() construct may only be used in the source of a mapping rule and may not be used in a bidirectional mapping rule.

$t($and(**classExpr**$_1$ ... **classExpr**$_n$)$, X) \mapsto$
$t($**classExpr**$_1, X)$ and ... and $t($**classExpr**$_n, X)$

$t($or(**classExpr**$_1$ ... **classExpr**$_n$)$, X) \mapsto$
$t($**classExpr**$_1, X)$ or ... or $t($**classExpr**$_n, X)$

$t($not(**classExpr**)$, X) \mapsto$ 'naf' $t($**classExpr**$, X)$

$t($join(**classExpr**$_1$ ... **classExpr**$_n$ **logicalExpression**$_1$ ...
logicalExpression$_n$)) $\mapsto$
$t($**classExpr**$_1, f(?x_2, ..., ?x_n))$ 'impliedBy' $t($**classExpr**$_2, ?x_2)$ 'and' ... 'and'
$t($**classExpr**$_n, ?x_n)$ 'and' $t($**logicalExpression**$_1, \{f(?x_2, ..., ?x_n), ?x_2, ..., ?x_n\})$
'and' ... 'and' $t($**logicalExpression**$_1, \{f(?x_2, ..., ?x_n), ?x_2, ..., ?x_n\})$ '.'

One more transformation function is required, for the case when the **classExpr** is a **classID**:

$t($**classID**$, X) \mapsto X$ 'memberOf' $t($**classID**$)$

A.2 Attribute mappings

Below the translations of attribute mappings are specified. For the bidirectional attribute mapping we distinguish three cases: (1) no variables are given as parameters, (2) one variable is given and (3) two variables are given.

$t($*attributeMapping*$($*bidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$)) \mapsto$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$))$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_2$ attributeExpr$_{12}$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$))$

$t($*attributeMapping*$($*bidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X) \mapsto$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X)$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_2$ attributeExpr$_{12}$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X)$

$t($*attributeMapping*$($*bidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X, Y) \mapsto$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X, Y)$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_2$ attributeExpr$_{12}$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X, Y)$

$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$)) \mapsto$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), x_{new})$

$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X) \mapsto$
$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_m$$), X, x_{new})$

$t($*attributeMapping*$($*unidirectional* attributeExpr$_1$ attributeExpr$_2$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpression$_1$ ... logicalExpression$_m$$), X, Y) \mapsto$
$t($attributeExpr$_1, X, Y)$ 'impliedBy' $t($attributeExpr$_1, X, Y)$
'and' $t($attributeCondition$_1$, X, $t($attributeID$_1$$)$ 'and' ... 'and'
$t($attributeCondition$_n, X, t($attributeID$_1$$))$
'and' $t($logicalExpression$_1, X, Y)$ 'and' ... 'and'
$t($logicalExpression$_m, X, Y)$

The mappings for **attributeExpr**s implies the usage of a transformation function having three parameters: the attribute expression to be transformed, the **ID** of an instance of the concept owning this attribute, and the value of the attribute.

$t($**attributeID**$, X, Y) \mapsto X[$ $t($attributeID$)$ 'hasValue' $Y]$

$t($inverse(**attributeExpr**)$, X, Y) \mapsto$
$t($**attributeExpr**$, Y, X)$

$t($symmetric(**attributeExpr**)$, X, Y) \mapsto$
$t($**attributeExpr**$, X, Y)$ 'and' $t($**attributeExpr**$, Y, X)$

$t($reflexive(**attributeExpr**)$, X, Y) \mapsto$
$t($**attributeExpr**$, X, Y)$ 'and' $t($**attributeExpr**$, X, X)$

$t($trans(**attributeExpr**)$, X, Y) \mapsto t($**attributeExpr**$, X, Y)$ 'impliedBy'
$t($**attributeExpr**$, X, ?z)$ 'and' $t($**attributeExpr**$, ?z, Y)$

$t($and(**attributeExpr**$_1 \ldots$ **attributeExpr**$_n$)$, X, Y) \mapsto$
$t($**attributeExpr**$_1, X, Y)$ 'and' $\ldots$ 'and' $t($**attributeExpr**$_n, X, Y)$

$t($or(**attributeExpr**$_1 \ldots$ **attributeExpr**$_n$)$, X, Y) \mapsto$
$t($**attributeExpr**$_1, X, Y)$ 'or' $\ldots$ 'or' $t($**attributeExpr**$_n, X, Y)$

$t($not(**attributeExpr**$, X, Y) \mapsto$
'naf' $t($**attributeExpr**$, X, Y)$

The transformation function for **classCondition**s and **attributeCondition**s have the following definitions ($attID$ is a meta-identifier which is replaced with the actual attribute identifier during translation):

$t($attributeValueCondition(**attributeID**, **individualID**), $X) \mapsto$
X[$t($attributeID$)$ 'hasValue' $t($individualID))]

$t($attributeValueCondition(**attributeID**, **dataLiteral**), $X) \mapsto$
X[$t($**attributeID**$)$ 'hasValue' $t($**dataLiteral**))]

$t($attributeValueCondition(**attributeID**, **classExpr**)$, X) \mapsto$
X[$t($**attributeID**$)$ 'hasValue' $?y$] **and** $t($**classExpr**, $y)$

$t($attributeOccurenceCondition(**attributeID**)$, X) \mapsto$
$X[t($**attributeID**$)$ 'hasValue' $?x_{new}$]

$t($valueCondition(**individualID**)$, X, attID) \mapsto$
$X[attID$ 'hasValue' $t($**individualID**$)]$

$t($valueCondition(**dataLiteral**)$, X, attID) \mapsto$
$X[attID$ 'hasValue' $t($**dataLiteral**$)]$

$t($valueCondition(**classExpr**$, X, attID) \mapsto$
$X[attID$ 'hasValue' $?y$] 'and' $t($**classExpr**, $?y)$

$t($expressionCondition(**attributeExpr**), $X, attID) \mapsto$
$t($attributeExpr, $X, attID)$

Having defined the transformations of expressions and conditions in WSML we can start defining the transformations for the actual mappings.

A.3 Relation mappings

Below the translations of relation mappings into WSML-Rule are specified. A bidirectional relation mapping is translated to two unidirectional mappings:

$t($*relationMapping*$($*bidirectional* relationExpr$_1$ relationExpr$_2$
relationCondition$_1$ ... relationCondition$_m$
logicalExpression$_1$ ... logicalExpression$_n$$))\mapsto$
$t($*relationMapping*$($*unidirectional* relationExpr$_1$ relationExpr$_2$
relationCondition$_1$ ... relationCondition$_m$
logicalExpression$_1$ ... logicalExpression$_n$$))$
$t($*relationMapping*$($*unidirectional* relationExpr$_2$ relationExpr$_1$
relationCondition$_1$ ... relationCondition$_m$
logicalExpression$_1$ ... logicalExpression$_n$$))$

For each relation mapping, n new variables $(?x_1, ..., ?x_n)$ are introduced, where n is the arity of the relations. Notice that all relations in a relation mapping must have the same arity.

$t($*relationMapping*$($*unidirectional* relationExpr$_1$ relationExpr$_2$
relationCondition$_1$ ... relationCondition$_m$
logicalExpresion$_1$ ... logicalExpresion$_q$$))\mapsto$
$t($relationExpr$_1, ?x_1, ..., ?x_n)$ **impliedBy** $t($relationExpr$_2, ?x_1, ..., ?x_n))$ 'and'
$t($relationCondition$_1, ?x_1, ..., ?x_n, t($relationID$_1))$ 'and' ... 'and'
$t($relationCondition$_m, ?x_1, ..., ?x_n, t($relationID$_m))$ 'and'
$t($logicalExpression$_1, ?x_1, ..., ?x_n)$ 'and' ... 'and'
$t($logicalExpression$_n, ?x_1, ..., ?x_n)$ '.'

The transformation functions for **relationExpr** and **relationCondition**s are the followings ($relID$ is a meta-identifier which is replaced with the actual attribute identifier during translation):

$t($and(relationExpr$_1$, ..., relationExpr$_n$)$, ?x_1, ..., ?x_n)\mapsto$
$t($relationExpr$_1, ?x_1, ..., ?x_n)$ 'and' ... 'and' $t($relationExpr$_1, ?x_1, ..., ?x_n)$

$t($or(relationExpr$_1$, ..., relationExpr$_n$)$, ?x_1, ..., ?x_n)\mapsto$
$t($relationExpr$_1, ?x_1, ..., ?x_n)$ 'or' ... 'or' $t($relationExpr$_1, ?x_1, ..., ?x_n)$

$t($not(relationExpr)$, ?x_1, ..., ?x_n)\mapsto$ 'naf' $t($relationExpr, $?x_1, ..., ?x_n)$

$t($relationID$, ?x_1, ..., ?x_n)\mapsto$ relationID$(?x_1, ..., ?x_n)$

$t($parameterCondition(**individualID**)$, ?x_1, ..., ?x_k, ..., ?x_n,$ relID$)\mapsto$
relID$(?x_1, ..., ?x_k, ..., ?x_n)$ 'and' $?x_k$ = $t($**individualID**$)$

t(parameterCondition(**dataLiteral**), $?x_1, ..., ?x_k, ..., ?x_n$, relID) $\mapsto$
relID($?x_1, ..., ?x_k, ..., ?x_n$) 'and' $?x_k$ = t(**dataLiteral**)

t(parameterCondition(**classExpr**), $?x_1, ..., ?x_k, ..., ?x_n$, relID) $\mapsto$
relID($?x_1, ..., ?x_k, ..., ?x_n$) 'and' t(**classExpr**, $?x_k$)

t(expressionCondition(**relationExpr**), $?x_1, ..., ?x_n$, relID) $\mapsto$
t(relID, $?x_1, ..., ?x_n$)

A.4 Instance mappings

Instance mappings are not allowed in WSML-Rule, because equality in the head is not allowed in WSML-Rule.

A.5 Class-attribute mappings

Below the translations of class-attribute mappings into WSML-Rule are specified. Again, bidirectional mappings are translated into two unidirectional mappings.

t(*classAttributeMapping*(*bidirectional* classExpr attributeExpr
classAttributeMapping$_1$ ... classAttributeMapping$_n$
attributeMapping$_1$ ... attributeMapping$_m$
classCondition$_1$ ... classCondition$_p$
attributeCondition$_1$ ... attributeCondition$_q$
logicalExpresion$_1$ ... logicalExpresion$_s$)) $\mapsto$
t(*classAttributeMapping*(*unidirectional* classExpr attributeExpr
classAttributeMapping$_1$ ... classAttributeMapping$_n$
attributeMapping$_1$ ... attributeMapping$_m$
classCondition$_1$ ... classCondition$_p$
attributeCondition$_1$ ... attributeCondition$_q$
logicalExpresion$_1$ ... logicalExpresion$_s$))
t(*classAttributeMapping*(*unidirectional* attributeExpr classExpr
classAttributeMapping$_1$ ... classAttributeMapping$_n$
attributeMapping$_1$ ... attributeMapping$_m$
classCondition$_1$ ... classCondition$_p$
attributeCondition$_1$ ... attributeCondition$_q$
logicalExpresion$_1$ ... logicalExpresion$_s$))

$t($*classAttributeMapping*(*unidirectional* classExpr attributeExpr
classAttributeMapping$_1$ ... classAttributeMapping$_n$
attributeMapping$_1$ ... attributeMapping$_m$
classCondition$_1$ ... classCondition$_p$
attributeCondition$_1$ ... attributeCondition$_q$
logicalExpresion$_1$ ... logicalExpresion$_s$)) $\mapsto$
$(t($**attributeExpr**$, f(?x), ?y)$ 'impliedBy' $t($classExpr$, ?x)$ 'and
$t($classCondition$_1, ?x)$ 'and' ... 'and' $t($classCondition$_p, ?x)$ 'and'
$t($attributeCondition$_1, ?x)$ 'and' ... 'and' $t($attributeCondition$_n, ?x)$ 'and'
logicalExpresion$_1$ 'and' ... 'and' logicalExpresion$_s$) 'and'
$t($classAttributeMapping$_1, ?x, f(?x))$ 'and' ... 'and'
$t($classAttributeMapping$_n, ?x, f(?x))$ 'and'
$t($attributeMapping$_1, ?x, f(?x))$ 'and' ... 'and'
$t($attributeMapping$_m, ?x, f(?x))$ '.'

$t($*classAttributeMapping*(*unidirectional* attributeExpr classExpr
classAttributeMapping$_1$ ... classAttributeMapping$_n$
attributeMapping$_1$ ... attributeMapping$_m$
classCondition$_1$ ... classCondition$_p$
attributeCondition$_1$ ... attributeCondition$_n$
logicalExpresion$_1$ ... logicalExpresion$_s$)) $\mapsto$
$(t($classExpr$, f(?x))$ 'impliedBy' $t($**attributeExpr**$, ?x, ?y)$ 'and
$t($classCondition$_1, ?x)$ 'and' ... 'and' $t($classCondition$_p, ?x)$ 'and'
$t($attributeCondition$_1, ?x)$ 'and' ... 'and' $t($attributeCondition$_n, ?x)$ 'and'
logicalExpresion$_1$ 'and' ... 'and' logicalExpresion$_s$) 'and'
$t($classAttributeMapping$_1, ?x, f(?x))$ 'and' ... 'and'
$t($classAttributeMapping$_n, ?x, f(?x))$ 'and'
$t($attributeMapping$_1, ?x, f(?x))$ 'and' ... 'and'
$t($attributeMapping$_m, ?x, f(?x))$ '.'

Appendix B

OWL syntax

For expressing mappings between ontologies in OWL we rely on the ontology import mechanism of OWL, which is expressed through annotation properties in the OWL abstract syntax (in case **mappingID** is missing, it is simply omitted from the resulting OWL ontology).

t('MappingDocument(' **mappingID** 'source(' **ontologyID**$_1$ ')' $\ldots$ 'source(' **ontologyID**$_{n-1}$ ')' 'target(' **ontologyID**$_n$ ')' **directive**$_1$ $\ldots$ **directive**$_n$ ')') $\mapsto$
'Ontology(' t(**mappingID**) 'Annotation(owl:imports' t(**ontologyID**$_1$) ')' $\ldots$ 'Annotation(owl:imports' t(**ontologyID**$_n$) ')' t(**directive**$_1$) $\ldots$ t(**directive**$_n$) ')'

A directive is either an annotation or a mapping expression. We translate annotations in the following way:

t('Annotation(' **propertyID** **propertyValue** ')') $\mapsto$ 'Ontology(' t(**propertyID**) t(**propertyValue**) ')'

A mapping expression can be a class mapping, an attribute mapping, a relation mapping, an instance mapping or a mapping between these.

Note that a class mapping which omits the 'bidirectional' is equivalent to a class mapping which has the 'unidirectional' integrated.

t('classMapping(' ['unidirectional'] **classExpr**$_1$ **classExpr**$_2$ **classAttributeMapping**$_1$ $\ldots$ **classAttributeMapping**$_n$ **classCondition**$_1$ $\ldots$ **classCondition**$_n$ ['{' **logicalExpression** '}'] ')') $\mapsto$
'SubClassOf(' 'intersectionOf(' t(**classExpr**$_1$) t(**classAttributeMapping**$_1$) $\ldots$ t(**classAttributeMapping**$_n$) t(**classCondition**$_1$) $\ldots$ t(**classCondition**$_n$) ')' t(**classExpr**$_2$) ')' t(logicalExpression)

A bidirectional mapping simple translated to a class equivalence axiom:

t('classMapping(bidirectional' **classExpr**$_1$ **classExpr**$_2$ **classAttributeMapping**$_1$ $\ldots$ **classAttributeMapping**$_n$ **classCondition**$_1$ $\ldots$ **classCondition**$_n$ ['{' **logicalExpression**

'}'] ')') $\mapsto$
'EquivalentClasses(' 'intersectionOf(' $t($**classExpr**$_1)$ $t($**classAttributeMapping**$_1)$ $\ldots$ $t($**classAttributeMapping**$_n)$ $t($**classCondition**$_1)$ $\ldots$ $t($**classCondition**$_n)$ ')' $t($**classExpr**$_2)$ ')' $t($logicalExpression$)$

There exist two kinds of attribute mappings, namely attribute mappings inside class mappings and attribute mappings separate of class mappings. Because of limitations in OWL, attributes can only be mapped outside of the context of the class. Furthermore, attribute conditions are not allowed, because they cannot be applied in property axioms in OWL.

$t($'attributeMapping(' ['unidirectional'] **attributeExpr**$_1$ **attributeExpr**$_2$ ['{' **logicalExpression** '}'] ')'$)$ $\mapsto$
'SubPropertyOf(' $t($**attributeExpr**$_1)$ $t($**attributeExpr**$_2)$ $t($logicalExpression$)$

$t($'attributeMapping(bidirectional' **attributeExpr**$_1$ **attributeExpr**$_2$ ['{' **logicalExpression** '}'] ')'$)$ $\mapsto$
'EquivalentProperties(' $t($**attributeExpr**$_1)$ $t($**attributeExpr**$_2)$ $t($logicalExpression$)$

When mapping between OWL ontologies, attributes are equivalent to relations, because OWL only has the concept of binary relations:

$t($'relationMapping(' ['unidirectional'] **relationExpr**$_1$ **relationExpr**$_2$ ['{' **logicalExpression** '}'] ')'$)$ $\mapsto$
'SubPropertyOf(' $t($**relationExpr**$_1)$ $t($**relationExpr**$_2)$ $t($logicalExpression$)$

$t($'relationMapping(bidirectional' **relationExpr**$_1$ **relationExpr**$_2$ ['{' **logicalExpression** '}'] ')'$)$ $\mapsto$
'SubPropertyOf(' $t($**relationExpr**$_1)$ $t($**relationExpr**$_2)$ $t($logicalExpression$)$

An instance mapping is simply equivalent to individual equality in OWL:

$t($'instanceMapping(' **individualID**$_1$ **individualID**$_2$ ')' $)$ $\mapsto$
'SameIndividual(' $t($**individualID**$_1)$ $t($**individualID**$_2)$

Appendix C

First-Order Reference Semantics

This appendix contains a First-Order Logic (FOL) reference semantics for the mapping language described in this deliverable, in order to clarify the intention of the mappings. We have chosen FOL for this purpose, because it can be used to illustrate all aspects of the language.

Note that in the first-order reference semantics of the language, any first-order formula can be used in the place of a **logicalExpression**.

In the remainder of this appendix, we define the mapping function t, which takes as argument a mapping specified in the mapping language and which returns a set of first-order formulas.

It is possible to use a number of meta-variables in logical expressions which are nested inside other mapping expression (for example, class mappings). The meta-variables $X_1, ..., X_n$ and are syntactically substituted in the translation of the mapping language to FOL:

$t($**logicalExpression**$, Y_1, ..., Y_n) \mapsto$
logicalExpression$[X_1 := Y_1, ..., X_n := Y_n]$

Below, the translations of class mappings are specified. First, a bidirectional class mapping is translated into two unidirectional class mappings. Then, a unidirectional mapping is translated as a rule of subtranslations.

$t($classMapping(bidirectional **classExpr**$_1$ **classExpr**$_2$
  attributeMapping$_1$ ... **attributeMapping**$_n$
  classCondition$_1$ ... **classCondition**$_m$
  logicalExpression$_1$ ... **logicalExpression**$_q$)) $\mapsto$
$t($classMapping(unidirectional **classExpr**$_1$ **classExpr**$_2$
  attributeMapping$_1$ ... **attributeMapping**$_n$
  classCondition$_1$ ... **classCondition**$_m$

logicalExpression$_1$ ... **logicalExpression**$_q$))
$t($classMapping(unidirectional **classExpr**$_2$ **classExpr**$_1$
attributeMapping$_1$ ... **attributeMapping**$_n$
classCondition$_1$ ... **classCondition**$_m$
logicalExpression$_1$ ... **logicalExpression**$_q$))

$t($classMapping(unidirectional **classExpr**$_1$ **classExpr**$_2$
attributeMapping$_1$ ... **attributeMapping**$_n$
classCondition$_1$ ... **classCondition**$_m$
logicalExpression$_1$ ... **logicalExpression**$_q$)) $\mapsto$
$t($**classExpr**$_1, x) \rightarrow t($**classExpr**$_2, x) \wedge$
$t($**attributeMapping**$_1, x) \wedge \ldots \wedge t($**attributeMapping**$_n, x)$
$t($**classCondition**$_1, x) \wedge \ldots \wedge t($**classCondition**$_m, x) \wedge$
$t($**logicalExpression**$_1, x) \wedge ... \wedge t($**logicalExpression**$_q, x)$.

In the mapping language, for different class expressions different translations are required. There are no explicit constructs for representing the intersection, union, difference and join operations in WSML. Therefore, we have to create a new concept and to write for it the WSML logical expression that defines the intersection, union, complement, and join, respectively. Note that the or() construct may only be used in the source of a mapping rule and may not be used in a bidirectional mapping rule.

$t($and(**classExpr**$_1$ ... **classExpr**$_n$)$, X) \mapsto$
$t($**classExpr**$_1, X) \wedge ... \wedge t($**classExpr**$_n, X)$

$t($or(**classExpr**$_1$ ... **classExpr**$_n$)$, X) \mapsto$
$t($**classExpr**$_1, X) \vee ... \vee t($**classExpr**$_n, X)$

$t($not(**classExpr**)$, X) \mapsto \neg t($**classExpr**$, X)$

$t($join(**classExpr**$_1$ ... **classExpr**$_n$ **logicalExpression**$_1$ ...
logicalExpression$_n$)) $\mapsto$
$t($**classExpr**$_1, f(x_2, ..., x_n)) \leftarrow t($**classExpr**$_2, x_2) \wedge ... \wedge$
$t($**classExpr**$_n, x_n) \wedge t($**logicalExpression**$_1, \{f(x_2, ..., x_n), x_2, ..., x_n\}) \wedge ... \wedge$
$t($**logicalExpression**$_1, \{f(x_2, ..., x_n), x_2, ..., x_n\})$ '.'

One more transformation function is required, for the case when the **classExpr** is a **classID** (a simple class identifier):

$t($**classID**$, X) \mapsto$ **classID**(X)

Below the translations of attribute mappings are specified. For the bidirectional attribute mapping we distinguish three cases: (1) no variables are given as parameters, (2) one variable is given and (3) two variables are given.

$t($attributeMapping(bidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$)) \mapsto$
$t($attributeMapping(unidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$))$
$t($attributeMapping(unidirectional **attributeExpr**$_2$ **attributeExpr**$_1$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$))$

$t($attributeMapping(bidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$), X) \mapsto$
$t($attributeMapping(unidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$), X)$
$t($attributeMapping(unidirectional **attributeExpr**$_2$ **attributeExpr**$_1$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$), X)$

$t($attributeMapping(bidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$), X, Y) \mapsto$
$t($attributeMapping(unidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$), X, Y)$
$t($attributeMapping(unidirectional **attributeExpr**$_2$ **attributeExpr**$_1$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$), X, Y)$

$t($attributeMapping(unidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$)) \mapsto$
$t($attributeMapping(*unidirectional* **attributeExpr**$_1$ **attributeExpr**$_2$
  attributeCondition$_1$ ... **attributeCondition**$_n$
  logicalExpression$_1$ ... **logicalExpression**$_m$$), x_{new})$

$t($attributeMapping(unidirectional **attributeExpr**$_1$ **attributeExpr**$_2$

attributeCondition$_1$ ... **attributeCondition**$_n$
logicalExpression$_1$ ... **logicalExpression**$_m$$), X) \mapsto$
$t($attributeMapping(unidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
attributeCondition$_1$ ... **attributeCondition**$_n$
logicalExpression$_1$ ... **logicalExpression**$_m$$), X, x_{new})$

$t($attributeMapping(unidirectional **attributeExpr**$_1$ **attributeExpr**$_2$
attributeCondition$_1$ ... **attributeCondition**$_n$
logicalExpression$_1$ ... **logicalExpression**$_m$$), X, Y) \mapsto$
$t($**attributeExpr**$_2, X, Y) \leftarrow t($**attributeExpr**$_1, X, Y) \wedge$
$t($**attributeCondition**$_1, X, t($**attributeID**$_1) \wedge ... \wedge$
$t($**attributeCondition**$_n, X, t($**attributeID**$_1)) \wedge$
$t($**logicalExpression**$_1, X, Y) \wedge ... \wedge$
$t($**logicalExpression**$_m, X, Y)$

$t($**attributeID**$, X, Y) \mapsto$ **attributeID**(X, Y)

$t($inverse(**attributeExpr**)$, X, Y) \mapsto$
$t($**attributeExpr**$, Y, X)$

$t($symmetric(**attributeExpr**)$, X, Y) \mapsto$
$t($**attributeExpr**$, X, Y) \wedge t($**attributeExpr**$, Y, X)$

$t($reflexive(**attributeExpr**)$, X, Y) \mapsto$
$t($**attributeExpr**$, X, Y) \wedge t($**attributeExpr**$, X, X)$

$t($trans(**attributeExpr**)$, X, Y) \mapsto t($**attributeExpr**$, X, Y) \leftarrow$
$t($**attributeExpr**$, X, z) \wedge t($**attributeExpr**$, z, Y)$

$t($and(**attributeExpr**$_1 \ldots$ **attributeExpr**$_n$)$, X, Y) \mapsto$
$t($**attributeExpr**$_1, X, Y) \wedge \ldots \wedge t($**attributeExpr**$_n, X, Y)$

$t($or(**attributeExpr**$_1 \ldots$ **attributeExpr**$_n$)$, X, Y) \mapsto$
$t($**attributeExpr**$_1, X, Y) \vee ... \vee t($**attributeExpr**$_n, X, Y)$

$t($not(**attributeExpr**$, X, Y) \mapsto$
$\neg t($**attributeExpr**$, X, Y)$

The transformation function for **classCondition**s and **attributeCondition**s have the following definitions ($attID$ is a meta-identifier which is replaced with the actual attribute identifier during translation):

$t($attributeValueCondition(**attributeID**, **individualID**), $X) \mapsto$
attributeID(X, **individualID**)

$t($attributeValueCondition(**attributeID**, **dataLiteral**), $X) \mapsto$
attributeID(X,**dataLiteral**)

$t($attributeValueCondition(**attributeID**, **classExpr**), $X) \mapsto$
$\exists y($**attributeID**$(X, y) \wedge t($**classExpr**, $y))$

$t($attributeOccurenceCondition(**attributeID**), $X) \mapsto$
$\exists y($**attributeID**$(X, y))$

$t($valueCondition(**individualID**)$, X, attID) \mapsto$
$attID(X,$**individualID**$)$

$t($valueCondition(**dataLiteral**)$, X, attID) \mapsto$
$attID(X,$**dataLiteral**$)$

$t($valueCondition(**classExpr**$, X, attID) \mapsto$
$\exists y(attID(X, y) \wedge t($**classExpr**, $y))$

$t($expressionCondition(**attributeExpr**)$, X, attID) \mapsto$
$t($attributeExpr, $X, attID)$

Having defined the transformations of expressions and conditions in we can start defining the transformations for the actual mappings.

$t($relationMapping(bidirectional **relationExpr**$_1$ **relationExpr**$_2$
relationCondition$_1$ ... **relationCondition**$_m$
logicalExpression$_1$ ... **logicalExpression**$_n$$)) \mapsto$
$t($relationMapping(unidirectional **relationExpr**$_1$ **relationExpr**$_2$
relationCondition$_1$ ... **relationCondition**$_m$
logicalExpression$_1$ ... **logicalExpression**$_n$$))$
$t($relationMapping(unidirectional **relationExpr**$_2$ **relationExpr**$_1$
relationCondition$_1$ ... **relationCondition**$_m$
logicalExpression$_1$ ... **logicalExpression**$_n$$))$

For each relation mapping, n new variables $(x_1, ..., x_n)$ are introduced, where n is the arity of the relations. Notice that all relations in a relation mapping must have the same arity.

t(relationMapping(unidirectional **relationExpr**$_1$ **relationExpr**$_2$
  relationCondition$_1$ ... **relationCondition**$_m$
  logicalExpression$_1$ ... **logicalExpression**$_q$)) $\mapsto$
$t(\textbf{relationExpr}_1, x_1, ..., x_n) \leftarrow t(\textbf{relationExpr}_2, x_1, ..., x_n))$ 'and'
  $t(\textbf{relationCondition}_1, x_1, ..., x_n, t(\textbf{relationID}_1)) \wedge ...\wedge$
  $t(\textbf{relationCondition}_m, x_1, ..., x_n, t(\textbf{relationID}_m))\wedge$
  $t(\textbf{logicalExpression}_1, x_1, ..., x_n) \wedge ...\wedge$
  $t(\textbf{logicalExpression}_n, x_1, ..., x_n)$ '.'

The transformation functions for **relationExpr** and **relationCondition**s are the followings ($relID$ is a meta-identifier which is replaced with the actual attribute identifier during translation):

t(and(**relationExpr**$_1$, ..., **relationExpr**$_n$)$, X_1, ..., X_n) \mapsto$
$t(\textbf{relationExpr}_1, X_1, ..., X_n) \wedge ... \wedge t(\textbf{relationExpr}_1, X_1, ..., X_n)$

t(**or(relationExpr**$_1$, ..., **relationExpr**$_n$)$, X_1, ..., X_n) \mapsto$
$t(\textbf{relationExpr}_1, X_1, ..., X_n) \vee ... \vee t(\textbf{relationExpr}_1, X_1, ..., X_n)$

$t(\textbf{not(relationExpr)}, X_1, ..., X_n) \mapsto \neg t(\textbf{relationExpr}, X_1, ..., X_n)$

$t(\textbf{relationID}, X_1, ..., X_n) \mapsto \textbf{relationID}(X_1, ..., X_n)$

t('instanceMapping(' **individualID**$_1$ **individualID**$_2$ ')') $\mapsto$
individualID$_1$=**individualID**$_2$

t(classAttributeMapping(bidirectional **classExpr attributeExpr**
  classAttributeMapping$_1$ ... **classAttributeMapping**$_n$
  attributeMapping$_1$ ... **attributeMapping**$_m$
  classCondition$_1$ ... **classCondition**$_p$
  attributeCondition$_1$ ... **attributeCondition**$_q$
  logicalExpression$_1$ ... **logicalExpression**$_s$)) $\mapsto$
t(classAttributeMapping(unidirectional **classExpr attributeExpr**
  classAttributeMapping$_1$ ... **classAttributeMapping**$_n$
  attributeMapping$_1$ ... **attributeMapping**$_m$
  classCondition$_1$ ... **classCondition**$_p$
  attributeCondition$_1$ ... **attributeCondition**$_q$

logicalExpression$_1$ ... **logicalExpression**$_s$))
$t($classAttributeMapping(unidirectional **attributeExpr classExpr**
classAttributeMapping$_1$ ... **classAttributeMapping**$_n$
attributeMapping$_1$ ... **attributeMapping**$_m$
classCondition$_1$ ... **classCondition**$_p$
attributeCondition$_1$ ... **attributeCondition**$_q$
logicalExpression$_1$ ... **logicalExpression**$_s$))

$t($classAttributeMapping(unidirectional **classExpr attributeExpr**
classAttributeMapping$_1$ ... **classAttributeMapping**$_n$
attributeMapping$_1$ ... **attributeMapping**$_m$
classCondition$_1$ ... **classCondition**$_p$
attributeCondition$_1$ ... **attributeCondition**$_q$
logicalExpression$_1$ ... **logicalExpression**$_s$)) $\mapsto$
$(t($**attributeExpr**$, f(x), y) \leftarrow t($**classExpr**$, x)\wedge$
$t($**classCondition**$_1, x) \wedge ... \wedge t($**classCondition**$_p, x)\wedge$
$t($**attributeCondition**$_1, x) \wedge ... \wedge t($**attributeCondition**$_n, x)\wedge$
logicalExpression$_1$ $\wedge ... \wedge$ **logicalExpression**$_s)\wedge$
$t($**classAttributeMapping**$_1, x, f(x)) \wedge ...\wedge$
$t($**classAttributeMapping**$_n, x, f(x))\wedge$
$t($**attributeMapping**$_1, x, f(x)) \wedge ...\wedge$
$t($**attributeMapping**$_m, x, f(x))$.

$t($classAttributeMapping(unidirectional **attributeExpr classExpr**
classAttributeMapping$_1$ ... **classAttributeMapping**$_n$
attributeMapping$_1$ ... **attributeMapping**$_m$
classCondition$_1$ ... **classCondition**$_p$
attributeCondition$_1$ ... **attributeCondition**$_n$
logicalExpression$_1$ ... **logicalExpression**$_s$)) $\mapsto$
$(t($**classExpr**$, f(x)) \leftarrow t($**attributeExpr**$, x, y)\wedge$
$t($**classCondition**$_1, x) \wedge ... \wedge t($**classCondition**$_p, x)\wedge$
$t($**attributeCondition**$_1, x) \wedge ... \wedge t($**attributeCondition**$_n, x)\wedge$
logicalExpression$_1$ $\wedge ... \wedge$ **logicalExpression**$_s)\wedge$
$t($**classAttributeMapping**$_1, x, f(x)) \wedge ...\wedge$
$t($**classAttributeMapping**$_n, x, f(x))\wedge$
$t($**attributeMapping**$_1, x, f(x)) \wedge ...\wedge$
$t($**attributeMapping**$_m, x, f(x))$.

Appendix D

SableCC Grammar

This chapter presents the grammar of the mapping language in the sablecc format. Sablecc[1] stands for sable compiler compiler. This tool generates a parser given the following grammar file. We use this parser in the mapping language API.

The grammar is specified using a dialect of Extended BNF which can be used directly in the SableCC compiler compiler. Terminals are delimited with single quotes, non-terminals are underlined and refer to the corresponding productions. Alternatives are separated using vertical bars ’|’; optional elements are appended with a question mark ’?’; elements that may occur zero or more times are appended with an asterisk ’*’; elements that may occur one or more times are appended with a plus ’+’. In the case of multiple references to the same non-terminal in a production, the non-terminals are disambiguated by using labels of the form ’[label]:’.

D.1 Helpers

```
all  =  [ 0x0 .. 0xffff ]
escape_char  =  '\'
basechar  =  [ 0x0041 .. 0x005A ] | [ 0x0061 .. 0x007A ]
ideographic  =  [ 0x4E00 .. 0x9FA5 ] | 0x3007 | [ 0x3021 .. 0x3029 ]
letter  =  basechar | ideographic
digit  =  [ 0x0030 .. 0x0039 ]
combiningchar  =  [ 0x0300 .. 0x0345 ] | [ 0x0360 .. 0x0361 ]
| [ 0x0483 .. 0x0486 ]
extender  =  0x00B7 | 0x02D0 | 0x02D1 | 0x0387 | 0x0640
| 0x0E46 | 0x0EC6 | 0x3005 | [ 0x3031 .. 0x3035 ] |
 [ 0x309D .. 0x309E ] | [ 0x30FC .. 0x30FE ]
alphanum  =  digit | letter
```

[1]http://www.sablecc.com

```
hexdigit  =  [ '0' .. '9' ] | [ 'A' .. 'F' ]
not_escaped_ncnamechar  =  letter | digit | '_' | combiningchar | extender
escaped_ncnamechar  =  '.' | '-' | not_escaped_ncnamechar
ncnamechar  =  ( escape_char escaped_ncnamechar ) | not_escaped_ncnamechar
reserved  =  '/' | '?' | '#' | '[' | ']' | ';'
| ':' | '@' | '&'
| '=' | '+' | '$' | ','
mark  =  '-' | '_' | '.' | '!' | '~' | '*' | ''' | '(' | ')'
escaped  =  '%' hexdigit hexdigit
unreserved  =  letter | digit | mark
scheme  =  letter ( letter | digit | '+' | '-' | '.' )*
port  =  digit*
idomainlabel  =  alphanum ( ( alphanum | '-' )* alphanum )?
dec_octet  =  digit | ( [ 0x31 .. 0x39 ] digit ) | ( '1' digit digit )
| ( '2' [ 0x30 .. 0x34 ] digit ) | ( '25' [ 0x30 .. 0x35 ] )
ipv4address  =  dec_octet '.' dec_octet '.' dec_octet '.' dec_octet
h4  =  hexdigit hexdigit? hexdigit? hexdigit?
ls32  =  ( h4 ':' h4 ) | ipv4address
ipv6address  =  ( ( h4 ':' )* h4 )? '::' ( h4 ':' )* ls32
| ( ( h4 ':' )* h4 )? '::' h4 | ( ( h4 ':' )* h4 )? '::'
ipv6reference  =  '[' ipv6address ']'
ucschar  =  [ 0xA0 .. 0xD7FF ] | [ 0xF900 .. 0xFDCF ]
| [ 0xFDF0 .. 0xFFEF ]
iunreserved  =  unreserved | ucschar
ipchar  =  iunreserved | escaped | ';' | ':' | '@' | '&' | '='
| '+' | '$' | ','
isegment  =  ipchar*
ipath_segments  =  isegment ( '/' isegment )*
iuserinfo  =  ( iunreserved | escaped | ';' | ':' | '&' | '='
| '+' | '$' | ',' )*
iqualified  =  ( '.' idomainlabel )* '.' ?
ihostname  =  idomainlabel iqualified
ihost  =  ( ipv6reference | ipv4address | ihostname )?
iauthority  =  ( iuserinfo '@' )? ihost ( ':' port )?
iabs_path  =  '/' ipath_segments
inet_path  =  '//' iauthority ( iabs_path )?
irel_path  =  ipath_segments
ihier_part  =  inet_path | iabs_path | irel_path
iprivate  =  [ 0xE000 .. 0xF8FF ]
iquery  =  ( ipchar | iprivate | '/' | '?' )*
ifragment  =  ( ipchar | '/' | '?' )*
iri_f  =  scheme ':' ihier_part ( '?' iquery )? ( '#' ifragment )?
absolute_iri  =  scheme ':' ihier_part ( '?' iquery )?
relative_iri  =  ihier_part ( '?' iquery )? ( '#' ifragment )?
iric  =  reserved | iunreserved | escaped
iri_reference  =  iri_f | relative_iri
```

```
tab  =  9
cr  =  13
lf  =  10
eol  =  cr lf | cr | lf
squote  =  '''
dquote  =  '"'
not_cr_lf  =  [ all - [ cr+ lf ] ]
escaped_char  =  escape_char all
not_escape_char_not_dquote  =  [ all - [ '"' + escape_char ] ]
not_escape_char_not_squote  =  [ all - [ ''' + escape_char ] ]
literal_content  =  escaped_char | not_escape_char_not_dquote
string_content  =  escaped_char | not_escape_char_not_squote
not_star  =  [ all - '*' ]
not_star_slash  =  [ not_star - '/' ]
long_comment  =  '/*' not_star* '*' +
( not_star_slash not_star* '*' + )* '/'
begin_comment  =  '//' | 'comment'
short_comment  =  begin_comment not_cr_lf* eol
comment  =  short_comment | long_comment
blank  =  ( ' ' | tab | eol )+
qmark  =  '?'
luridel  =  '<"'
ruridel  =  '">'
primary_subtag  =  letter+
language_subtag  =  ( letter | digit )+
language_tag  =  '@' primary_subtag ( '-' language_subtag )*
```

D.2 Tokens

```
t_blank  =  blank
t_comment  =  comment
comma  =  ','
endpoint  =  '.' blank
pathcon  =  '.'
dblcaret  =  '^^'
lpar  =  '('
rpar  =  ')'
lbracket  =  '['
rbracket  =  ']'
lbrace  =  '{'
rbrace  =  '}'
colon  =  ':'
gt  =  '>'
```

```
lt  =  '<'
gte  =  '>='
lte  =  '=<'
equals  =  '='
unequal  =  '!='
add_op  =  '+'
sub_op  =  '-'
star  =  '*'
div_op  =  '/'
t_mappingdocument  =  'MappingDocument'
source  =  'source'
target  =  'target'
t_annotation  =  'annotation'
t_measure  =  'measure'
unidirectional  =  'unidirectional'
bidirectional  =  'bidirectional'
classmapping  =  'classMapping'
attributemapping  =  'attributeMapping'
relationmapping  =  'relationMapping'
instancemapping  =  'instanceMapping'
classattributemapping  =  'classAttributeMapping'
classrelationmapping  =  'classRelationMapping'
classinstancemapping  =  'classInstanceMapping'
attributeclassmapping  =  'attributeClassMapping'
relationclassmapping  =  'relationClassMapping'
instanceclassmapping  =  'instanceClassMapping'
t_transformation  =  'transformation'
service  =  'service'
and  =  'and'
or  =  'or'
not  =  'not'
join  =  'join'
inverse  =  'inverse'
symetric  =  'symetric'
transitive  =  'trans'
reflexive  =  'reflexive'
univ_false  =  'false'
univ_true  =  'true'
attributevaluecondition  =  'attributeValueCondition'
attributeoccurencecondition  =  'attributeOccurenceCondition'
valuecondition  =  'valueCondition'
expressioncondition  =  'expressionCondition'
string  =  squote string_content* squote
plainliteral  =  dquote literal_content* dquote language_tag?
full_iri  =  luridel iri_reference ruridel
ncname  =  ( letter | '_' ) ncnamechar*
```

```
anonymous  =  '_#' digit*
pos_int  =  digit+
pos_float  =  digit+ '.' digit+
Ignored Tokens

    * t_blank
    * t_comment
```

D.3 Productions

```
mappingdocument  =
t_mappingdocument lpar documentid source_exp
target_exp annotation* expression* rpar
source_exp  =
source lpar ontologyid rpar
target_exp  =
target lpar ontologyid rpar
annotation  =
t_annotation lpar propertyid propertyvalue rpar
measure  =
t_measure lpar float rpar
expression  =
{logical_expression}  lbrace annotation* logicalexpression rbrace
|  {class_mapping} classmapping lpar annotation* measure? directionality?
[first]: classexpr [second]: classexpr
classcondition* logicalexprbrace? rpar
|  {attribute_mapping} attributemapping lpar annotation*
measure? directionality?
[first]: attributeexpr [second]: attributeexpr
attributecondition* transformation? logicalexprbrace? rpar
|  {relation_mapping} relationmapping lpar annotation*
measure? directionality?
[first]: relationexpr [second]: relationexpr
relationcondition* logicalexprbrace? rpar
|  {instance_mapping} lpar annotation*
[first]: instanceid [second]: instanceid rpar
|  {classattribute_mapping} classattributemapping lpar
annotation* measure? directionality?
[first]: classexpr [second]: attributeexpr
logicalexprbrace? rpar
|  {classrelation_mapping} classrelationmapping lpar
annotation* measure? directionality?
[first]: classexpr [second]: relationexpr
```

```
logicalexprbrace? rpar
|  {classinstance_mapping} classinstancemapping lpar
annotation* measure? directionality?
[first]: classexpr [second]: instanceid logicalexprbrace? rpar
|  {attributeclass_mapping} attributeclassmapping lpar
annotation* measure? directionality?
[first]: attributeexpr [second]: classexpr logicalexprbrace? rpar
|  {relationclass_mapping} relationclassmapping lpar
annotation* measure? directionality?
[first]: relationexpr [second]: classexpr logicalexprbrace? rpar
|  {instanceclass_mapping} instanceclassmapping lpar
annotation* measure? directionality?
[first]: instanceid [second]: classexpr logicalexprbrace? rpar
classexpr  =
{classid}  classid
|  {and}  and lpar [first]: classexpr [second]: classexpr classexpr* rpar
|  {or}  or lpar [first]: classexpr [second]: classexpr classexpr* rpar
|  {not}  not lpar classexpr rpar
attributeexpr  =
{attributeid}  attributeid
|  {and}  and lpar [first]: attributeexpr [second]: attributeexpr
attributeexpr* rpar
|  {or}  or lpar [first]: attributeexpr [second]: attributeexpr
attributeexpr* rpar
|  {not}  not lpar attributeexpr rpar
|  {inverse}  inverse lpar attributeexpr rpar
|  {symetric}  symetric lpar attributeexpr rpar
|  {reflexive}  reflexive lpar attributeexpr rpar
|  {transitive}  transitive lpar attributeexpr rpar
|  {join}  join lpar [first]: attributeexpr [second]: attributeexpr
attributeexpr* logicalexprbrace? rpar
relationexpr  =
{relationid}  relationid arity?
|  {and}  and lpar [first]: relationexpr [second]: relationexpr
relationexpr* rpar
|  {or}  or lpar [first]: relationexpr [second]: relationexpr
relationexpr* rpar
|  {not}  not lpar relationexpr rpar
|  {join}  join lpar [first]: relationexpr [second]: relationexpr
relationexpr* logicalexprbrace? rpar
arity  =
lbracket arity_val rbracket
classcondition  =
attributevaluecondition lpar attributeid
indidordatalittorclassexpr rpar
relationcondition  =
string
attributecondition  =
expressioncondition lpar attributeexpr rpar
transformation  =
{function}  t_transformation lpar functionid param* rpar
|  {service}  t_transformation lpar service iri param* rpar
iri  =
full_iri
id  =
{iri}  iri
|  {anonymous}  anonymous
|  {literal}  literal
|  {universal_truth}  univ_true
|  {universal_falsehood}  univ_false
prefix  =
ncname colon
lexicalform  =
plainliteral
literal  =
{typedliteral}  typedliteral
|  {plainliteral}  plainliteral
|  {numeric}  number
|  {string}  string
typedliteral  =
plainliteral dblcaret iri
neg_int  =
sub_op pos_int
float  =
sub_op? pos_float
number  =
{positive_int}  pos_int
|  {negative_int}  neg_int
|  {float}  float
logicalexpression  =
string
logicalexprbrace  =
lbrace logicalexpression rbrace
directionality  =
{unidirectional}  unidirectional
|  {bidirectional}  bidirectional
indidordatalittorclassexpr  =
iri
confidence  =
pos_float
relationaloperator  =
{greaterthan}  gt
```

```
|  {lowerthan}  lt
|  {equals}  equals
arity_val  =
pos_int
propertyvalue  =
string
documentid  =
iri
ontologyid  =
iri
classid  =
iri
propertyid  =
iri
attributeid  =
iri
relationid  =
iri
instanceid  =
iri
functionid  =
{string}  string
|  {iri}  iri
param  =
{string}  string
|  {iri}  iri
|  {number}  number
```