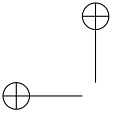
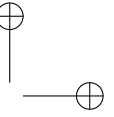
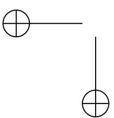
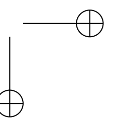
Roma Tre University
Ph.D. in Computer Science and Engineering

Interoperability of Semantic Annotations

Stefano Paolozzi

Interoperability of Semantic Annotations

A thesis presented by
Stefano Paolozzi
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Engineering

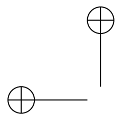
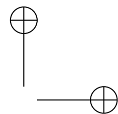
Roma Tre University
Dept. of Informatics and Automation

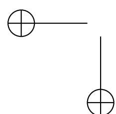
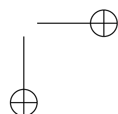
April 2009

Committee:
Prof. Paolo Atzeni

Reviewers:
Prof. Silvana Castano
Prof. Esperanza Marcos

To Alessandra, whose strength has always supported me in this PhD adventure.

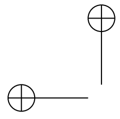
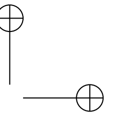
Abstract

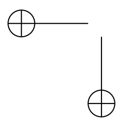
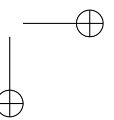
The Semantic Web is the new generation World Wide Web. It extends the Web by giving information a well defined meaning, allowing it to be processed by machines. This vision is going to become reality thanks to a set of technologies which have been specified and maintained by the World Wide Web Consortium (W3C), and more and more research efforts from the industry and the academia. Therefore, the basis for the Semantic Web are computer-understandable descriptions of resources. We can create such descriptions by annotating resources with metadata, resulting in "annotations" about that resource. Semantic annotation is the creation of metadata and relations between them with the task of defining new methods of access to information and enriching the potentialities of the ones already existent. The main goal is to have information on the Web, defined in such a way that its meaning could be explicitly interpreted also by automatic systems, not just by humans.

There is huge amount of interesting and important information represented through semantic annotations, but there are still a lot of different formalisms showing a lack of standardization and a consequent need of interoperability.

This growing need of interoperability in this field convinces us to extend our first proposal, strictly related to database models, in order to address also semantic annotations. Our proposal, mainly based on Model Management techniques, focuses on the problem of translating schemas and data between Semantic Web data models and the integration of those models with databases models that are a more rigid and well-defined structure.

In this work we underline the main concepts of our approach discussing a proposal for the implementation of the model management operator ModelGen, which translates schemas from one model to another focusing on semantic annotation context. The approach expresses the translation as Datalog rules and exposes the source and target of the translation in a generic relational dictionary. This makes the translation transparent, easy to customize and model-independent. The proposal includes automatic generation of translations as composition of basic steps.

Acknowledgments

I would like to express my deep gratitude to everyone who helped me shape the ideas explored in this dissertation, either by giving technical advice or encouraging and supporting my work in many other ways.

I am grateful to Prof. Paolo Atzeni who supervised and guided my work. He gave me the opportunity to conduct this doctoral research and helped me make the right decisions.

I would like to thank my colleagues and friends in the Database Groups at Roma TRE University, in particular to Giorgio and Piero who participated in this adventure with me.

Finally, I owe very special thanks to my family.

Contents

List of Figures **xii**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Contribution 4
1.3 Organization of the Thesis 4

2 Semantic Annotations **7**
2.1 Semantic Annotation - Why 8
2.2 Semantic Annotation - How 9
2.3 Semantic Annotations Issues 12
2.4 Discussion 12

3 The Model Management Approach **15**
3.1 Model Management 15
3.2 Model Independent Schema and Data Translation 18
3.3 Discussion 24

4 Motivating Scenarios **25**
4.1 Interoperability for Semantic Annotation Platforms 25
4.2 NOMOS vs NITE XML Toolkit 28
4.3 RDF vs Topic Maps 35
4.4 Discussion 41

5 Bridging the Gap between Semantic Annotations and Databases **43**
5.1 Motivation 43
5.2 Related Work 46
5.3 Discussion 47

6 OWL and Relational Database Mappings **49**
- 6.1 Towards Ontology and Databases Integration 50
- 6.2 An Extended Supermodel 60
- 6.3 From OWL Ontologies to Relational Databases 65
- 6.4 From Relational Databases to OWL ontologies 77
- 6.5 Information Loss 84
- 6.6 Discussion 85

7 A Tool Supporting Semantic Annotation Interoperability **87**
- 7.1 The MIDST Tool 87
- 7.2 Experimental Results 92
- 7.3 Discussion 93

Conclusion **95**

Appendices **97**

A. Datalog Rules for RDB to OWL translations **99**

B. Datalog Rules for OWL to RDB translations **111**

Bibliography **137**

List of Figures

1.1 Correspondences between Model Management and Database terminologies. . . . 3
1.2 Structural organization of the thesis. . . . 6

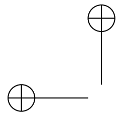
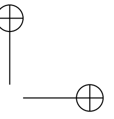
3.1 A simplified conceptual view of models and constructs. . . . 19
3.2 The relational implementation of the dictionary (portion). . . . 20
3.3 A simple Datalog rule. . . . 23

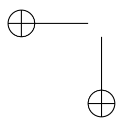
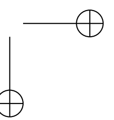
4.1 Classification of Semantic Annotation Platforms. . . . 26
4.2 The NOMOS Model. . . . 28
4.3 The NXT Model. . . . 29
4.4 The Nomos Metamodel. . . . 30
4.5 The NXT Metamodel. . . . 31
4.6 Correspondence between NOMOS and the supermodel. . . . 33
4.7 Correspondence between NXT and the supermodel. . . . 33
4.8 A simple translation from NXT to NOMOS. . . . 34
4.9 A portion of the supermodel showing the RDF-related metacostructs. 38
4.10 A portion of the supermodel showing the TM-related metacostructs. 39
4.11 A simple RDF graph. . . . 39
4.12 Topic Maps example. . . . 41
4.13 Resulting RDF graph. . . . 41

6.1 Relational Metamodel. . . . 51
6.2 Relational Metamodel in terms of Supermodel metaconstructs. . . 52
6.3 The Semantic Web layer cake. . . . 54
6.4 The main constructs of OWL Lite. . . . 55
6.5 OWL Lite metamodel. . . . 58
6.6 OWL Restriction for RelationshipBetweenClasses construct. 59

6.7 Correspondences between OWL Lite elements and OWL data model constructs. 61
6.8 Correspondences between OWL Model and the Supermodel. 62
6.9 A portion of the extended Supermodel. 65
6.10 The translation process. 66
6.11 Dictionary Tables for the OWL Example. 69
6.12 A portion of the Supermodel dictionary. 70
6.13 The inheritance process. 72
6.14 Results of step 5. 74
6.15 Results of step 6. 75
6.16 Translation within the Supermodel. 76
6.17 The resulting relational dictionary. 78
6.18 The resulting relational schema. 79
6.19 The relational schema to be translated. 79
6.20 The "copy" of the relational schema in the Supermodel. 80
6.21 The result of relational schema translation in the Supermodel. . . 82
6.22 Dictionary tables of resulting OWL schema. 83

7.1 Creation of a new construct in a model. 89
7.2 The visualization of the OWL model. 89
7.3 Specification of an OWL restriction through the relationshipBetweenClasses construct. 90
7.4 The import/export tool. 91

Chapter 1

Introduction

"No matter what it is, there is nothing that cannot be done. If one manifests the determination, he can move heaven and earth as he pleases. But because man is pluckless, he cannot set his mind to it. Moving heaven and earth without putting forth effort is simply a matter of concentration."

Yamamoto Tsunetomo *The way of the Samurai* (1716)

1.1 Background and Motivation

Metadata is descriptive information about data and applications. Metadata is used to specify how data is represented, stored, and transformed, or may describe interfaces and behaviour of software components.

Metadata-related activities arise in data management, Web site and portal management, network management, and in various fields of computer-aided engineering.

Data integration is a main topic in several contexts. Solving a data integration problem requires the manipulation of metadata that describe the sources and targets of the integration and mappings between those schemas. Work on metadata problems goes back to at least the early 1970s, when data translation was a hot database research topic. However, until recently there was no widely-accepted conceptual framework for this field, as there is for many database topics.

The problem of translating schemas between data models is acquiring pro-

gressive significance in heterogeneous environments. Applications are usually designed to deal with information represented according to a specific data model, while the evolution of systems (in databases as well as in other technology domains, such as the Web) led to the adoption of many representation paradigms.

Heterogeneity arises because data sources are independently developed by different people and for different purposes and subsequently need to be integrated. The data sources may use different data models, different schemas, and different value encodings.

A typical example of such "rich" data sources are semantic annotations (SA). Semantic annotation is the creation of metadata and relations between them with the task of defining new methods of access to information and enriching the potentialities of the ones already existent. The main goal is to have information on the Web, defined in such a way that its meaning could be explicitly interpreted also by automatic systems, not just by human beings.

A key objective of data integration is to provide a uniform view covering a number of this kind of heterogeneous data sources.

Using such a view, the data that resides at the sources can be accessed in a uniform way. This data is usually described using database schemas, such as relational, or XML schemas. To construct a uniform view, source schemas are matched to identify their similarities and discrepancies. The relevant portions of schemas are extracted and integrated into a uniform schema. The *translation* of data from the representation used at the sources into the representation conforming to the uniform schema is specified using database transformations.

An interesting research area that has been recently exploited to address these issues is called *model management*.

A central concept in generic model management is that of a *model*. A model is a formal description of a metadata artifact. Examples of models include relational database models, ontologies, interface definitions, object diagrams, etc. The manipulation of models usually involves designing *transformations* between models. Formal descriptions of such transformations are called mappings. Examples of mappings are SQL views, XSL transformations, ontology articulations, mappings between class definitions and relational schemas, mappings between two versions of a model, etc. The key idea behind model management is to develop a set of algebraic operators that generalize the transformation operations utilized across various metadata applications. These operators are applied to models and mappings as a whole rather than to their individual elements, and simplify the programming of metadata applications.

We remark that in this dissertation we use the terms schema and data model

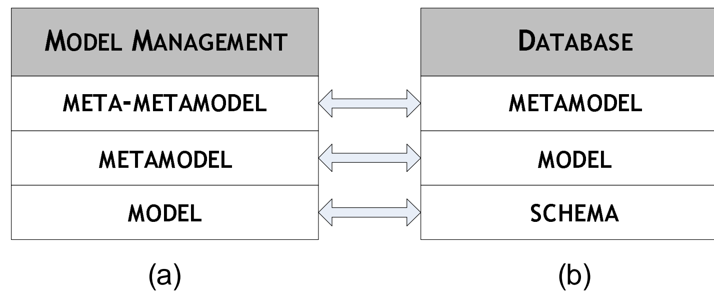


Figure 1.1: Correspondences between Model Management and Database terminologies.

as common in the database literature, though a recent trend in model management follows a different terminology (and uses model instead of schema and metamodel instead of data model [BM07] [BH07]). Figure 1.1 illustrates the correspondences between the recent model management terminology (Fig. 1.1a) and the one used in this dissertation (Fig. 1.1b).

An ambitious goal is to consider translations in a model generic setting [AT96] [Ber03], where the main problem can be formulated as follows: given two data models M_1 and M_2 (from a set of models of interest) and a schema S_1 of M_1, translate S_1 into a schema S_2 of M_2 that properly represents S_1.

Our first work was specifically oriented to database model transformations [ACB06] [ACB05]. However the never-ending spread of Semantic Web models is bringing new integration problems.

The Semantic Web comes from the idea of Tim Berners-Lee [BLHL01] that the Web as a whole can be made more intelligent and perhaps even intuitive about how to serve a users needs. Although search engines index much of the Web's content, they have little ability to select the pages that a user really wants or needs. Berners-Lee foresees a number of ways in which developers and authors, singly or in collaborations, can use self-descriptions and other techniques so that the context-understanding programs can selectively find what users want. The Semantic Web is specifically a Web of machine-readable information whose meaning is well-defined by standards: it absolutely needs the interoperable infrastructure that only global standard protocols can provide.

This need of interoperability convinces us to extend our first proposal, that was strictly related to database models, to address also Semantic Web models and in particular those models related to the wide research area of semantic

annotations.

This dissertation focuses on the problem of translating schemas and data between Semantic Web data models and the integration of those models with databases models that have a more rigid and well-defined structure.

1.2 Contribution

The main contributions of this thesis that is a part of a research project at Roma Tre University I have contributed are:

- the adoption of a general model to properly represent a broad range of data models. The proposed general model is based on the idea of *construct*: a construct represents a "structural" concept of a data model. We find out a construct for each "structural" concept of every considered data model and, hence, a data model can be completely represented by means of its constructs set.
- the extension of the general model approach to properly represent all (ideally) Semantic Web data models with particular attention to semantic annotation models. For each model, we explain how its concepts are represented by means of constructs and relationships between them. Several semantic annotation formalisms have been considered (RDF, Topic Maps, OWL, etc.) jointly with several SA platforms models.
- the definition of a set of brand new Datalog-like rules to perform the translation between semantic annotation models an the integration of those models and data in databases.
- the implementation of a flexible framework that allows to validate the concepts of the approach and to test their effectiveness. The main components of the tool, include a set of modules to support users in defining and managing models, schemas, Skolem functions, translations, import and export of schemas.

1.3 Organization of the Thesis

The thesis is organized into five main parts with several chapters each. Figure 1.2 shows the relations between these parts and chapters. The arrows indicate the suggested path for reading.

In *Part I: Background*, the first chapter introduces and motivates the presented research issues. Chapter 2.4 covers the needed background and basic concepts of semantic annotations.

Part II: Semantic Annotations Interoperability starts with Chapter 3.3 describing the basics of our approach, exploiting Model Management techniques. Chapter 4.4 illustrates a first application of our approach for semantic annotation interoperability at two different levels, namely the platform level and the language level. Chapters 5.3 and 6.6 cover the extension of the approach to manage the integration of semantic annotations, defined by means of ontologies, and databases. In these chapters we supply an in-depth description of our methodology.

Part III: Implementation presents the implemented tool to realize semantic annotation interoperability.

Part IV: Discussion points out the scientific contributions, concludes the work and discusses further research opportunities.

Part V: Appendices contains additional material like the mapping rules used to perform translation between different semantic annotation models.

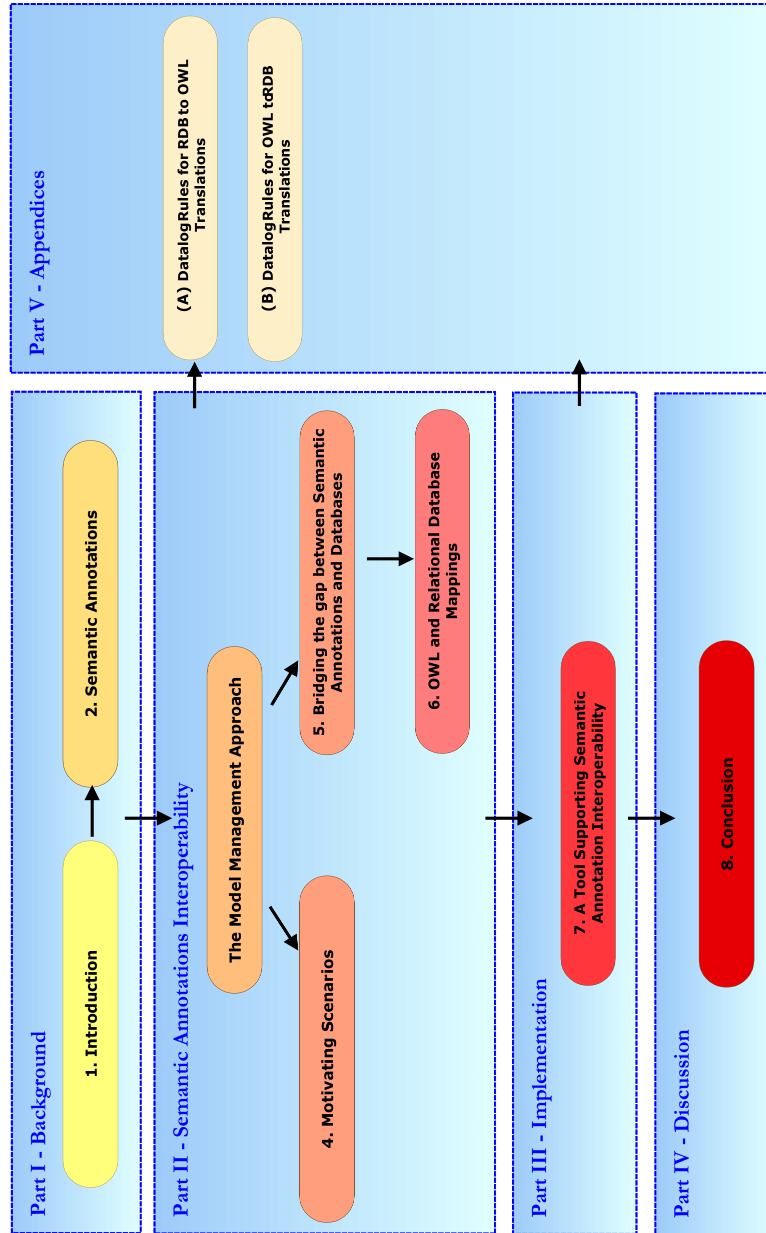


Figure 1.2: Structural organization of the thesis.

Chapter 2

Semantic Annotations

Semantic annotation can play a significant role in the broad area of the Semantic Web. Indeed, using metadata that describes the content of resources, is the way to perform operations, such as rich searching and retrieval, over those resources. In addition, if rich semantic metadata is supplied, those agents can then employ reasoning over the metadata, enhancing their processing power. The main idea of this approach is the provision of semantic annotations, both through automatic and human means. In this section we describe semantic annotations showing the main characteristics and underlining the open issues.

The Semantic Web vision, as proposed by Tim Berners-Lee [BL99], regards a Web in which resources are accessible not only to humans, but also to machines (e.g. computers, automated process roaming the Web, etc.). The automation of tasks depends on elevating the status of the Web from machine-readable to something we might call machine-understandable. The key idea is to have data on the Web defined and linked in such a way that its meaning is explicitly interpretable by software processes rather than just being implicitly interpretable by human beings [BCG+02].

To realize this vision, it will be necessary to associate metadata (i.e., data "about" data) with Web resources. One useful mechanism for associating such metadata is annotation. Indeed, most of the information currently available on the Web was produced to be used by people and it is therefore hardly suitable

for automatic processing, with "anonymous syntax" rather than "explicit semantic". Semantic annotation refers to the use of metadata for the description of Web resources, and the term is used to indicate both the metadata itself and the process that produces the metadata [Han05].

Semantic annotations are used to enrich the informative content of Web documents and to express in more formal way the meaning of a resource. The result is Web pages with machine interpretable mark-up that provide the source material with which agents and Semantic Web services operate. The goal is to create annotations with well-defined semantics, however those semantics may be defined.

2.1 Semantic Annotation - Why

The importance of semantic annotations relies on the capability to make the content of those annotations from human-readable to machine-readable. Consequently, applications can use those information and reason from them.

Manual annotation is more easily to perform, using available authoring tools for simultaneously authoring and annotating text. However, human annotation is often a complex and error-prone operation due to several factors, such as: i) annotator familiarity with the domain; ii) amount of training; iii) personal motivation and iv) complex schemas [BG03]. Manual annotation is also an expensive process, and often does not consider that multiple perspectives of a data source can be particularly useful to support the needs of different users. A typical example is a vision-impaired user who can use annotations to improve Web sites navigation [YHGS03]. Another problem with manual annotation is the huge amount of existing contents on the Web that must be annotated in order to be effectively used as a part of the Semantic Web. For this reason we can assert that manual semantic annotation has lead to a knowledge acquisition bottleneck [MS01].

To overcome the limitation of manual annotation, semi-automatic annotation of contents has been proposed. We refer to semi-automatic means, as opposed to completely automatic, because it is not possible yet to identify and classify all entities in source documents in a total automatic way. Indeed, all existing semantic annotation systems are based on human intervention during the annotation process, using the paradigm of balanced cooperative modeling [MS01]. Automated annotation provides the scalability needed to annotate existing documents on the Web and reduces the complexity in annotating new resources.

2.2 Semantic Annotation - How

A semantic annotation method is characterized by a number of features, such as the level of formality or the intended usage, that determine the kind of annotation that can be performed. We briefly illustrate six key features and, for each of them, examples of semantic annotation tools that cover such features are given.

Kind of Resource to be Annotated

Different annotation tools may be used to add semantics to various kinds of resources with the purpose of increasing their information content, for example: documents corpora or part of them, Web pages, multimedia contents (images, audio/video, etc.) or e-mails. One of the tool that supports such kind of annotation is MnM [VVMD$^+$02b].

Other kinds of resources that can be annotated are the Web services. In this case, input and output parameters, but also pre-conditions and effects that characterize a Web service could be annotated, to obtain their formal description. Examples of tools that allow to annotate Web services are: IRS II ([MDCG03]) and METEOR-S ([POSV04] [SGR08]), that also uses SAWSDL, the new W3C standard for describing Semantic Web Services [KVBF07].

Also data schemas can be annotated just like annotation of such structures can be used for addressing interoperability problems among heterogeneous software applications, in order to resolve their semantic clashes.

Target User

Annotations may be conceived for human users or processed by a machine. In a human-oriented semantic annotation, the content of the resource is generally represented in a descriptive way. On the other side, in a machine-oriented annotation, the conceptual content of the resource is generally represented in a formal way. For instance, Annotea provides a framework for human-oriented annotations [KKPS02] [SHN07], while AeroDAML [Kog01a] provides machine-oriented annotations.

Machine-oriented tools can be further categorized into two different categories:

- systems that need to be used by knowledge engineers (people that are aware of formal languages)

- systems that can be used by domain experts (people that are aware of the application domain)

Typical examples of the first category tools are: MnM, Cohse [BHS04] and SMORE [KGHP]. An example of tool for the second category is RAT-O [MKH03].

Level of Formality

An annotation attaches some data to some other data: it establishes, within some context, a (typed) relation between the annotated data and the annotating data.

We distinguish three levels of formality: informal, formal and ontological. Informal annotations are not machine-understandable because they do not use a formal language. Formal annotations are machine-understandable, but do not use ontological terms. In ontological annotations the terminology has a commonly understood meaning that corresponds to a shared conceptualization of a particular domain (i.e. an ontology [Gru95]).

An informal annotation can be a phrase expressed in natural or controlled language (such as CLCE [1]).

Formal annotations have formally defined constituents and are machine-readable. In this case an annotation can be defined with languages such as XML.

Ontological annotation consists in an expression of a knowledge representation language (e.g. RDF(S), DAML [2], DAML+OIL [3] or OWL [4]).

For instance, OntoMat-Annotizer [HSC02] [HSV03] and NOMOS [NG06] allow to define ontological annotations, while Annotea allows to define informal annotations.

Terminology Restriction

The language used to annotate a resource can be used in the following ways:

[1] Common Logic Controlled English (CLCE) is a formal language with an English-like syntax. Anyone who can read ordinary English can read sentences in CLCE with little or no training. Writing CLCE, however, requires practice in learning to stay within its syntactic and semantic limitations. Formally, CLCE supports full first-order logic with equality supplemented with an ontology for sets, sequences, and integers.

[2] DAML: http://www.daml.org/2000/10/daml-ont.html

[3] http://www.ontoknowledge.org/oil/

[4] OWL: Ontology Web Language, http://www.w3.org/TR/owl-features/

- *without restriction*: the terms that appear in the annotation expression can be arbitrarily chosen among all the terms of the natural language

- *suggested restriction*: a restricted set of terms to be used for the semantic annotation are submitted to the annotator. This set could be composed from the terms of a glossary, a taxonomy, or a domain ontology. The annotator has an additional possibility of inserting further natural language terms to express concepts not included in the proposed set.

- *strict restriction*: the annotator must use, in the composition of the semantic annotation expression, only terms included in a glossary, or in a reference ontology (for ontology-based annotation).

For example, Annotea allows to attach a free text annotation to a resource, while MnM, allows suggested language restriction. Finally, OntoMat-Annotizer and AeroDAML represent ontology-based annotations.

Position of Annotation

An annotation expression can be *embedded* in the body of the resource or *attached* to it.

In the first situation the annotator has to modify the original document in order to perform the annotation. In a different manner he can work on a local copy of the document, but he will lose the possibility to make his/her annotations available to other users.

If the annotation is attached, it is separately stored and linked to the annotated resource via a pointer. In this case one must keep the reference pointer always updated with the effective document location, to preserve a meaningful annotation. For instance, in MnM we have embedded annotations, while in Chose we have attached annotations.

Context

It indicates the context of the annotation: when it was made, by whom, and within what scope. The annotation could, for example, be temporally scoped (it is only valid in 2008) or even spatially scoped (it is only valid in Japan). If the annotation is not about a document, then the context could also be the document the annotation is derived from.

Granularity

This criteria (also called "scope" [SP05] or "lexical span" [MTP06]) illustrates the possibility to annotate a resource at different levels. Considering a textual document as an example of resource, it is possible to annotate the entire document, a phrase in the document or even a particular word.

2.3 Semantic Annotations Issues

The evidence of the increasing interest in semantic annotations is also the relevant number of tools developed in the last years. Several issues have been addressed in this context, mainly concerning:

- Efficiency
- Scalability
- Flexibility
- Interoperability

Remarkable importance is covered by semantic interoperability, because it introduces notable challenges. The semantic interoperability is, in general, the ability to share the "meaning" of available information and of the applications built on them. From the semantic annotations point of view, this opens the possibility of operating with heterogenous resources by providing a bridge of common techniques and methods.

However, several approaches exist to create and manage semantic annotations, but, currently there is no formal model that can capture all these approaches. The lack of such a model causes additional problems. Indeed it complicates the comparison and evaluation of tools and, more important, makes integration of annotations difficult.

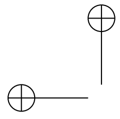
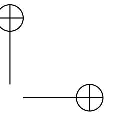
Our approach focuses on this problem defining a formal way to solve semantic interoperability for semantic annotations exploiting model management techniques.

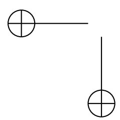
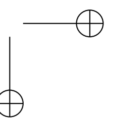
2.4 Discussion

Semantic annotation is information about which entities (or, more generally, semantic features) appear in a resource and where they do. Formally, semantic

annotations represent a specific sort of metadata, which provides references to entities in the form of URIs or other types of unique identifiers.

In this section we describe the fundamentals of semantic annotations also showing the main research directions. In the next sections we present the focus of this dissertation that is the study of interoperability and integration of semantic annotations.

Chapter 3

The Model Management Approach

Model management is a new approach to meta data management that offers a higher level programming interface than current techniques. The main abstractions are models (e.g., schemas, interface definitions) and mappings between models. It treats these abstractions as bulk objects and offers such operators as Match, Merge, Diff, Compose, Apply, and ModelGen. In this chapter we describe the principles of Model Management and how these techniques can be exploited to manage interoperability between heterogeneous models.

3.1 Model Management

Many information system problems involve the design, integration and maintenance of complex application artifacts, such as application programs, databases, Web sites, workflow scripts, formatted messages, and user interfaces. Engineers who perform this work use tools to manipulate formal descriptions, or models, of these artifacts, such as object diagrams, interface definitions, database schemas, Web site layouts, control flow diagrams, XML schemas and form definitions. This manipulation usually involves designing transformations between models, which in turn require an explicit representation of mappings, which describes how two models are related to each other.

To address these issues, Bernstein et al. [Ber03] [BHP00] [Mel04] outlined an environment to support development of metadata intensive applications in different domains. They called the framework *model management*.

Model management is a generic approach to solve problems of data programmability where precisely engineered mappings are required. Applications include data warehousing, e-commerce, object-to-relational wrappers, enterprise information integration, database portals, and report generators.

A first class citizen of this approach is the *model*. A model is a formal description of a metadata artifact. Examples of models include databases and XML schemas, interface specifications, object diagrams, UML-like diagrams, device models, form definitions and also ontologies. In this work we use a terminology that differs from the traditional one (which is adopted also by OMG [OMG05] [OMG06]), indeed we use a database-like terminology, where a *schema* is the description of the structure of the database and a *data model* (briefly, a model) is a set of constructs you are allowed to use to define your schemas; with this terminology, examples of models are the relational one or (any variation of) the ER one. Bernstein [Ber03] uses the term model for what we call schema and metamodel for what we call model. As a higher level is also needed in this framework, then we will have a metamodel as well, which he would call *metametamodel*.

We want to remark that in this dissertation we use the terms schema and data model as commonly used in the database literature, though a recent trend in model management follows a different terminology (and uses model instead of schema and metamodel instead of data model).

The manipulation of schemas usually involves designing transformations between them: formal descriptions of such transformations are called schema mappings or, more simply, mappings. Examples of mappings are SQL views, XQuery scripts, ontology articulations, mappings between class definitions and relational schemas, mappings between two versions of a model, mappings between a Web page and the underlying database and many others.

The main idea behind model management is to develop a set of algebraic operators that generalize the transformations across various metadata applications. These operators are applied to schemas and mappings as key elements, rather than to their individual elements, and are generic; they can be used for various problems and different kinds of metadata artifacts. The most important model management operators are:

- *Match* - which takes as input two schemas and (automatically) returns a mapping between them. The mapping identifies combinations of objects

in the input schemas that are either equal or similar, based on some externally provided definition of equality and similarity.

- *Compose* - which takes as input two mappings between schemas and returns a mapping that combines the two mappings.
- *Extract* - which takes as input a schema and a mapping and returns the subsets of the schema involved in the mapping.
- *Merge* - which takes as input two schemas and returns a schema that corresponds to their "union" and two mappings between the original schemas and the output schema. In other terms, the Merge operation returns a copy of all objects of the input schemas, except that objects of the input schemas that are collapsed into a single object in the output.
- *Diff* - which tales as input a schema and a mapping and returns the subset of the schema that does not participate in the mapping.
- *ModelGen* - which takes as input a schema for a source model and a target model and returns the translation of the schema into the target model.

Model management operators can be used for solving schema evolution, data integration and other complex problems using convenient programs executed by a model management system. A generic model management system may improve productivity for metadata-intensive application. Some of the most crucial questions on model management [BHJ+00] have been answered recently. For example, a formal semantics has been given to most of the aforementioned operators (e.g. Merge [PVM+02] [PB03], Compose [MH03] [FKPT05] [NBM07] [BGMN08], Match [MBR01] [PVM+02]), but some questions are still open. The most interesting researches are now related to the ModelGen operator. Indeed there are few implementations of a general ModelGen operator that are able to translate schemas between several data models and that also consider the transformation of the instances. The reason is due to the lack of rigorous semantics that explains what outputs should be produced given a schema with its constraints. Moreover, if we consider less structured data models such as the ones from Semantic Web (e.g. RDF, TopicMpas, OWL, etc.) these problems become even more complex. This is particularly due to the fact that, in these models, schema and instances coexist with no clear distinction between them.

3.2 Model Independent Schema and Data Translation

In this scenario, I have contributed to a large research project at Roma Tre University whose goal is to develop a tool supporting the transformation of schema and data among a large variety of formats and data models.

In this section we describe the fundamentals of our MIDST (Model Independent Schema and Data Translation) approach.

Our approach is based on the idea of a *metamodel*, defined as a set of constructs that can be used to define models, which are instances of the metamodel. This idea is based on Hull and Kings observation [HK87] that the constructs used in most known models can be expressed by a limited set of generic (i.e., model-independent) *metaconstructs*: lexical, abstract, aggregation, generalization, function. In fact, we define a metamodel by means of a set of generic metaconstructs [AT93]. Each model is defined by its constructs and the metaconstructs they refer to.

The various constructs are related to one another by means of references (for example, each attribute of an abstract has a reference to the abstract it belongs to) and have properties that specify details of interest (for example, for each attribute we specify whether it is part of the identifier and for each aggregation of abstracts we specify its cardinalities).

A major concept in our approach is the *supermodel*, a model that has constructs corresponding to all the metaconstructs known to the system. Thus, each model is a specialization of the supermodel and a schema in any model is also a schema in the supermodel, apart from the specific names used for constructs.

The supermodel gives us two interesting benefits. First of all, it acts like a "pivot" model, so that it is sufficient to have translations from each model to and from the supermodel, rather than translations for every couple of models. Therefore a linear, and not a quadratic, number of translations is needed. Indeed, since every schema in any model is also a schema of the supermodel (without considering constructs renaming), the only needed translations are those within the supermodel with the target model in mind. A translation is composed of (i) a "copy" (with construct renaming) from the source model into the supermodel; (ii) an actual transformation within the supermodel, whose output includes only constructs allowed in the target model; (iii) another copy (again with renaming into the target model). The second advantage is related to the fact that the supermodel emphasizes the common features of models. Therefore, if two source models share a construct, then their translations towards similar target models could share a portion of the translation as well.

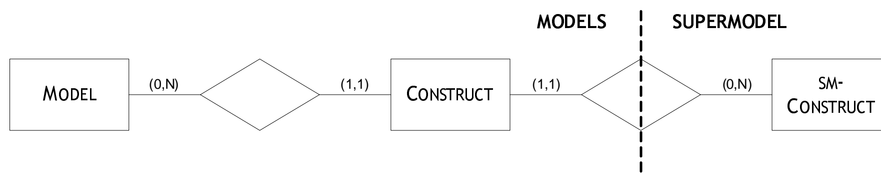


Figure 3.1: A simplified conceptual view of models and constructs.

In our approach, we follow this observation by defining elementary (or basic) translations that refer to single constructs (or even specific variants thereof). Then, actual translations are specified as compositions of basic ones, with significant reuse of them.

A conceptual view of the main elements of this idea is shown in Figure 3.1. The supermodel portion is predefined, but can be extended, whereas models are defined by specifying their respective constructs, where each of them refers to a construct of the supermodel (SM-CONSTRUCT) and so to a metaconstruct.

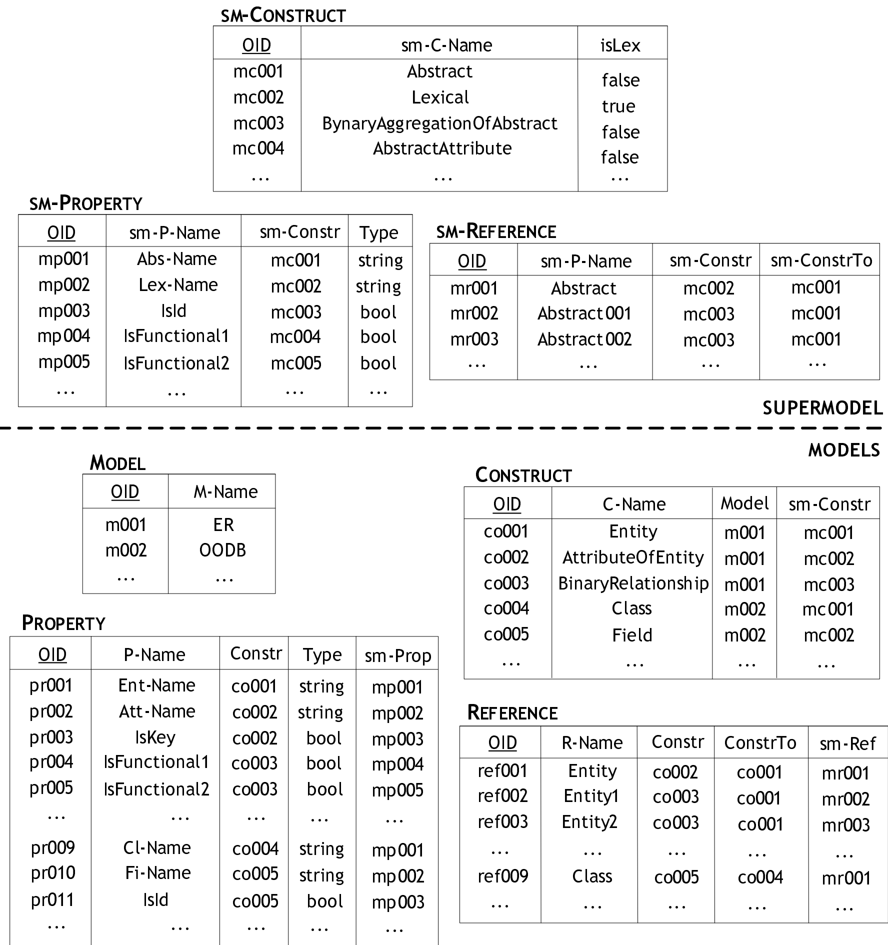
It is important to observe that our approach is independent of the specific supermodel that is adopted, as new metaconstructs and so SM-CONSTRUCTS can be added.

All the information about models and schemas is maintained in a dictionary that we have defined in a relational implementation. In Figure 3.2 we show the relational implementation of a portion of the dictionary that we used in our tool.

The actual implementation has more tables and more columns for each of them. We concentrate on the fundamentals omitting marginal details. The SM-CONSTRUCT table shows (a subset of) the generic constructs (which correspond to the metaconstructs). We have for example: ABSTRACT, LEXICAL, BINARYAGGREGATIONOFABSTRACTS, etc.

Each construct in the CONSTRUCT table refers to an SM-CONSTRUCT (by means of the SM-CONSTR column whose values contain foreign keys of SM-CONSTRUCT) and to a model (by means of MODEL). For example, the third row in table CONSTRUCT has value "mc003" for SM-CONSTR, in order to specify that "BinaryRelationship" is a construct (of the "ER" model, as indicated by value "m001" in the Model column) that refers to the "BinaryAggregationOfAbstract" SM-CONSTRUCT.

Tables SM-PROPERTY and SM-REFERENCE describe, at the supermodel level, the main features of constructs, properties and relationships among



SUPERMODEL

SM-CONSTRUCT

OID	sm-C-Name	isLex
mc001	Abstract	false
mc002	Lexical	true
mc003	BynaryAggregationOfAbstract	false
mc004	AbstractAttribute	false
...	...	...

SM-PROPERTY

OID	sm-P-Name	sm-Constr	Type
mp001	Abs-Name	mc001	string
mp002	Lex-Name	mc002	string
mp003	IsId	mc003	bool
mp004	IsFunctional1	mc004	bool
mp005	IsFunctional2	mc005	bool
...	...	...	...

SM-REFERENCE

OID	sm-P-Name	sm-Constr	sm-ConstrTo
mr001	Abstract	mc002	mc001
mr002	Abstract001	mc003	mc001
mr003	Abstract002	mc003	mc001
...	...	...	...

MODELS

MODEL

OID	M-Name
m001	ER
m002	OODB
...	...

CONSTRUCT

OID	C-Name	Model	sm-Constr
co001	Entity	m001	mc001
co002	AttributeOfEntity	m001	mc002
co003	BinaryRelationship	m001	mc003
co004	Class	m002	mc001
co005	Field	m002	mc002
...	...	...	...

PROPERTY

OID	P-Name	Constr	Type	sm-Prop
pr001	Ent-Name	co001	string	mp001
pr002	Att-Name	co002	string	mp002
pr003	IsKey	co002	bool	mp003
pr004	IsFunctional1	co003	bool	mp004
pr005	IsFunctional2	co003	bool	mp005
...	...	...	...	...
pr009	Cl-Name	co004	string	mp001
pr010	Fi-Name	co005	string	mp002
pr011	IsId	co005	bool	mp003
...	...	...	...	...

REFERENCE

OID	R-Name	Constr	ConstrTo	sm-Ref
ref001	Entity	co002	co001	mr001
ref002	Entity1	co003	co001	mr002
ref003	Entity2	co003	co001	mr003
...	...	...	...	...
ref009	Class	co005	co004	mr001
...	...	...	...	...

Figure 3.2: The relational implementation of the dictionary (portion).

them. Each SM-CONSTRUCT has some associated properties, described by SM-PROPERTY, which will then require values for each instance of each construct corresponding to it. For example, the second row of SM-PROPERTY describes the fact that each "Abstract" (mc002) has a "Lex-Name" of "string" type, whereas the third says that for each "Lexical" (mc002) we can specify whether it is part of the identifier of the "Abstract" or not (property "IsId"). Correspondingly, at the model level, we have that for each "AttributeOfEntity" we can tell whether it is part of the key (third row in table PROPERTY). In the latter case the property has a different name ("IsKey" rather than "IsId"). Other interesting properties are specified in the fourth and fifth rows of SM-PROPERTY. They allow for the specification of cardinalities of binary aggregations by saying whether the participation of an abstract is "functional" or not: a many-to-many relationship will have two false values, a one-to-one, two true ones and a one-to-many, a true and a false.

Table SM-REFERENCE describes how each SM-CONSTRUCT is related to another by specifying the references between them. For example, the second and the third row of SM-REFERENCE tells us that each "BinaryAggregationOfAbstract" (construct mc003) involves two "Abstract" constructs (mc001). Again, we have the issue repeated at the model level as well: the first row in table REFERENCE specifies that "AttributeOfEntity" (construct "c002", corresponding to the "Lexical" SM-CONSTRUCT) has a reference to "c001" ("Entity").

The supermodel part (in the upper part of Figure 3.2) is the core of the dictionary; it is predefined (but can be extended) and used as the basis for the definition of specific models. Essentially, the dictionary is initialized with the available SM-CONSTRUCT, with their properties and references. Initially, the model-specific part of the dictionary is empty and then individual models can be defined by specifying the constructs they include by referring to the SM-CONSTRUCT they refer to. In this way, the model part (in the lower part of Figure 3.2) is populated with rows that correspond to those in the supermodel part, except for the specific names, such as "Entity" or "AttributeOfEntity", which are model specific names for the SM-CONSTRUCTS "Abstract" and "Lexical" respectively. This structure causes some redundancy between the two portions of the dictionary, but this is not a great problem, as the model part is generated automatically: the definition of a model can be seen as a list of supermodel constructs, each with a specific name.

Basic Translations

In the current implementation of MIDST tool, translations are implemented by means of Datalog-variant rules with OID invention, where the latter feature is obtained through the use of Skolem functors [HY90]. This technique has several advantages:

- rules are independent of the main engine that interprets them, enabling rapid development of translations;
- the system itself can verify basic properties of sets of transformations (e.g., some form of correctness) by reasoning about the bodies and heads of Datalog rules [AGC08];
- transformations can be easily customized. For example, it is possible to add "selection condition" that specifies the schema elements to which a transformation is applied.

Another benefit of using Skolem functors is that their values can be stored in the dictionary and used to represent the mappings from a source to a target schema.

Each translation is usually concerned with a very specific task, such as eliminating a certain variant of a construct (possibly introducing another construct), with most of the constructs left unchanged. Therefore, in our programs only a few of the rules concerns real translations, whereas most of them just copy constructs from the source schema to the target one.

We use a non-positional notation for rules, so we indicate the names of the fields and omit those that are not needed (rather than using anonymous variables). Our rules generate constructs for a target schema (*tgt*) from those in a source schema (*src*). We may assume that variables *tgt* and *src* are bound to constants when the rule is executed. Each predicate has an OID argument. For each schema we have a different set of identifiers for the constructs. So, when a construct is produced by a rule, it has to have a "new" identifier. It is generated by means of a Skolem functor, denoted by the # sign in the rules. An example of rule is presented in Figure 3.3. This rule generates a new Abstract for each Abstract in the source schema.

We have the following restrictions on our rules. First, we have the standard "safety" requirements [UW97]: the literal in the head must have all fields, and each of them with a constant or a variable that appears in the body (in a positive literal) or a Skolem term. Similarly, all Skolem terms in the head or in the body have arguments that are constants or variables that appear in

ABSTRACT(
OID:#abstract_0(*absOid*),
Abs-Name: *name*
Schema: *tgt*)
←
ABSTRACT(
OID: *absOid*,
Abs-Name: *name*
Schema: *src*)

Figure 3.3: A simple Datalog rule.

the body. Moreover, our Datalog programs are assumed to be coherent with respect to referential constraints: if there is a rule that produces a construct *C* that refers to a construct *CO*, then there is another rule that generates a suitable *CO* that guarantees the satisfaction of the constraint.

Most of our rules are *recursive* according to the standard definition. However, recursion is only "apparent". A really recursive application happens only for rules that have atoms that refer to the target schema also in their body. In our experiments, we have developed a set of basic translations to handle the models that can be defined with our current metamodel.

More Complex Translations

Intuitively, complex translations can be performed by means of composition of basic rules as clearly described in [ACT+08].

However, with many possible models and many basic translations, it becomes important to understand how to find a suitable translation given a source and a target model. In this context we can find two main problems. The first one is how to verify what target model is generated by applying a basic step to a source model. The second problem is related to the "size" of the translations: due to the number of constructs and properties, we have too many models (a combinatorial explosion of them, if the variants of constructs grow) and it would be inefficient to find all associations between basic translations and pairs of models.

We propose a complete solution to the first issue, as follows. We associate

a concise description with each model, by indicating the constructs it involves with the associated properties (described in terms of propositional formulas), and a signature with each basic translation. Then, a notion of application of a signature to a model description allows us to obtain the description of the target model. With our basic translations written in a Datalog dialect with OID-invention, as we will see shortly, it turns out that signatures can be automatically generated and the application of signature gives an exact description of the target model.

With respect to the second issue, the complexity of the problem cannot be completely avoided, but we have defined algorithms that, under reasonable hypotheses, efficiently find a complex translation given a pair of models (source and target) (for more detail about complex translations see [ACT+08]).

3.3 Discussion

In this section we present the fundamentals of our approach to schema and data interoperability. We have exploited model management techniques in order to develop a tool that supports model-generic translation of schemas: given a source schema $S^{'}$ expressed in a source model and a target model TM, it generates a schema $S^{''}$ expressed in TM that is "equivalent" to $S^{'}$. The approach expresses the translation as Datalog rules and exposes the source and target of the translation in a generic relational dictionary. This makes the translation transparent, easy to customize and model-independent.

In the next sections we describe how this approach can be extended to manage the interoperability of Semantic Annotations and their integration in relational databases.

Chapter 4

Motivating Scenarios

Interoperability is the "ability of two or more systems or components to exchange information and to use the information that has been exchanged". In this section we describe two interesting interoperability contexts in which our approach can be successfully used. The first one regards interoperability at system levels, the second one is instead based on the language level.

The goal of semantic interoperability is to allow the (seamless) cooperation of two software applications that were not initially developed for this purpose. The cooperation will be possible without requiring the software applications to modify their software or their data organization. Semantic interoperability, in a broad vision, concerns process and information interoperability.

4.1 Interoperability for Semantic Annotation Platforms

The existence of various semantic annotation platforms (SAP) is interesting but, at the same time, gives rise to a difficulty, as the various tools have been developed independently from one another, and so it can become difficult to take advantage of annotations produced in different environments. Indeed, we have here an "interoperability" problem, similar to those we encounter in various information systems or database areas. Here interoperability refers to the possibility of sharing and exchanging annotations between different SAPs.

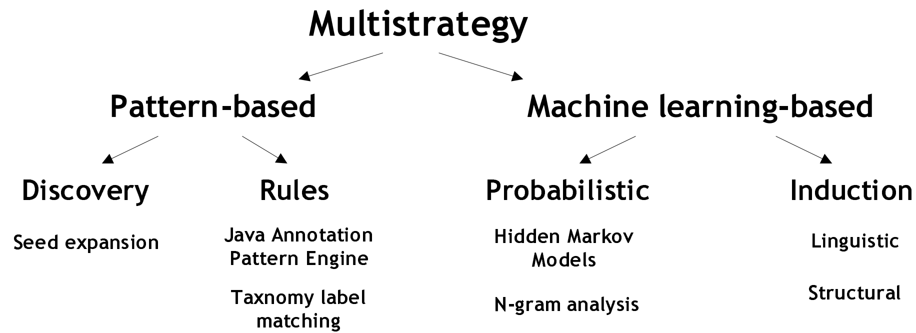


Figure 4.1: Classification of Semantic Annotation Platforms.

SAPs can be classified on the basis of the annotation method used. There are two primary categories, *Pattern-based* and *Machine Learning-based*, as shown in Figure 4.1. In addition, platforms can use methods from both types of categories, called *Multistrategy*, in order to take advantage of the strengths, and compensate for the weaknesses of the methods in each category.

In order to exploit our approach to SAPs, we have analyzed several applications that are related to current research projects. We present the most remarkable in the following.

AeroDAML

AeroDAML [Kog01b] is an annotation tool which applies information extraction techniques to automatically generate DAML annotations from Web pages. The aim is to provide naive users with a simple tool to create basic annotations without having to learn about ontologies, in order to reduce time and effort and to encourage people to semantically annotate their documents. AeroDAML links most proper nouns and common types of relations with classes and properties in a DAML ontology.

Amilcare

Amilcare [CW03] is a system which has been integrated in several different annotation tools for the Semantic Web. It uses machine learning to learn to adapt to new domains and applications using only a set of annotated texts

(training data). It has been adapted for use in the Semantic Web by simply monitoring the kinds of annotations produced by the user in training, and learning how to reproduce them. The traditional version of Amilcare adds XML annotations to documents (inline markup); the Semantic Web version leaves the original text unchanged and produces the extracted information as triples of the form $\langle annotation, startPosition, endPositioni \rangle$.

MnM

MnM [VvMD+02a] is a semantic annotation tool which provides support for annotating Web pages with semantic metadata. This support is semi-automatic, in that the user must provide some initial training information by manually annotating documents. It integrates a Web browser, an ontology editor, and it is Web-based and provides facilities for large-scale semantic annotation of Web pages.

NOMOS

NOMOS (aNnotation Of Media with Ontological Structure) is a platform for ontology-based semantic annotation of multimedia objects, developed at Stanford University Laboratories [NG06] [GNP]. Ontologies are specified in a formal way in OWL; this gives a rich meaning to the annotations and provides inference capabilities which enhance the annotation process as well as the annotations in many useful ways. NOMOS ontologies (or more precisely the terminological components of these ontologies) are used as high level schemas for annotations. In particular NOMOS has a reference ontology (called `corpora_2_0`) which is predefined, but can be extended. The reference ontology is based on abstract data definition such as: event, entity, relations, etc., which represent basic non-modifiable concepts.

NITE XML Toolkit

The NITE XML Toolkit (NXT) is an open source software tool for working with language corpora, within the meetings domain [JSUJ06]. It is designed to support the needs of human analysts as they work with a corpus, for tasks such as hand-annotation, automatic annotation where that relies on complex match patterns, data exploration, etc. NXT data model and query language are oriented toward those users who build descriptive analysis of heavily annotated multimodal corpora in preparation for defining appropriate statistical models.

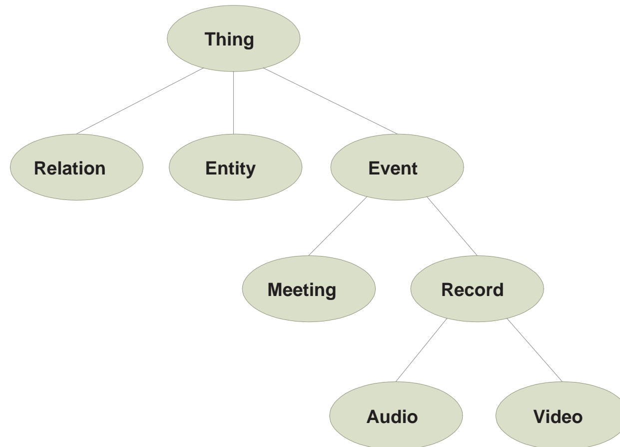


Figure 4.2: The NOMOS Model.

In NXT, annotations are described by types and attribute-value pairs. Moreover, they can be related to signals via start and end times, to representations of the external environment, and to each other via a graph structure. NXT represents the data for one meeting as a related set of XML files, with a metadata file that expresses information about the structure and location of the files. The main element of NXT data model is corpus. A corpus consists of a set of observations, each of which contains the data for one dialogue or interaction. Each of the dialogues or interaction types defined in a corpus involve a number of human or artificial agents. An agent is one actor in an observation. Each observation is recorded by one or more "signals" which consist of audio or video data characterizing the duration of the interaction.

4.2 NOMOS vs NITE XML Toolkit

In this section we will show how our approach can be used to handle translations between NOMOS and NXT which are clearly representative of the problems and allow us to illustrate the major features of our technique. The ontology concepts model of NOMOS is shown in Fig. 4.2, while the NXT model is shown in Figure 4.3 (the arrows indicate the reference between models constructs)

The data models of NOMOS and NXT have a number of similarities and

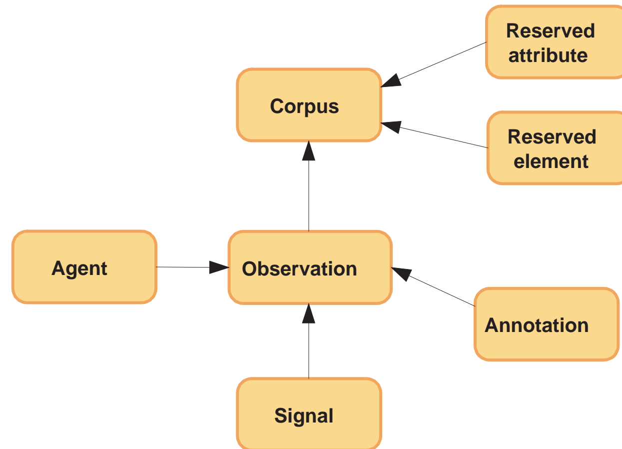


Figure 4.3: The NXT Model.

a few major differences. The most important similarity is that, despite some minor differences, it is possible to claim that the basic components of the two models, `Thing` in NOMOS and `Corpus` in NXT are very similar to each other. They correspond to notions we have in data models, at the instance level, such as occurrences of entities in the ER model or objects, belonging to classes, in object oriented models.

The way an annotation is described in terms of properties and related to one another is indeed slightly different in the two models. Indeed, NOMOS has an ontology concept to define the relation between things, whereas in NXT, relations between different corpora are not explicitly defined.

The actors of an annotation can be represented in both models. In NXT there is the `Agent` object, while in NOMOS we can use a specialization of the concept `Entity`, the only difference is on the categorization of the agents.

The `Event` concept in NOMOS ontology can reflect the `Observation` data object of NXT model, because they both represent a particular event of a meeting that can be observed.

A complete development of the analysis sketched above would allow us to understand which are the constructs in the two data models that are in close correspondence and which are unique to one of the two. In order to achieve this task it is useful to define a suitable metamodel (in ModelGen sense 3.2) of

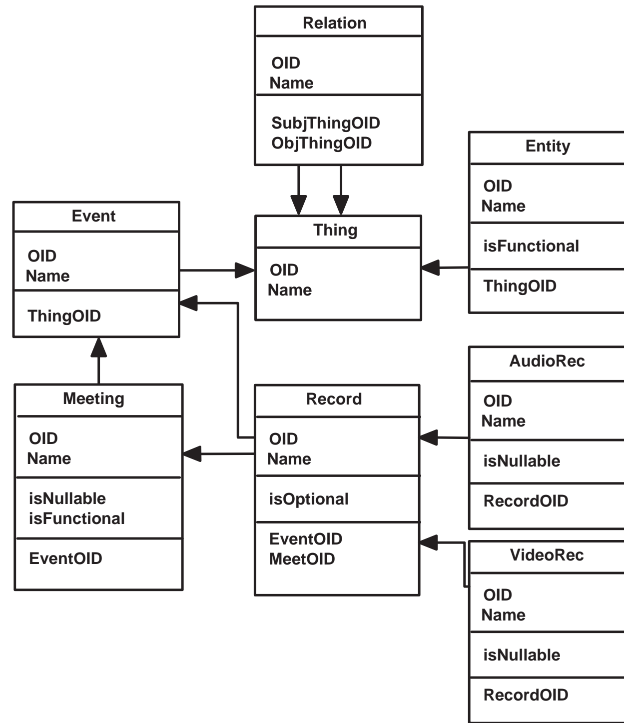


Figure 4.4: The Nomos Metamodel.

NOMOS and NXT. Possible metamodels are reported in Figures 4.4 and 4.5 where each construct is identified by a unique OID (Object Identifier). This would lead to the definition of a supermodel for this framework: a set of constructs, some of which appear (usually with different names) in both NOMOS and NXT models, and the others in only one.

The metamodel we use includes a few of the basic features, which we use essentially as representative of properties. For example minimum cardinalities are represented by means of the boolean IsOptional, so that allowed cardinalities are 0 ("true") and 1 ("false") and maximum cardinalities are represented by IsFunctional, with 1 and N as possible cardinality values. Many other features are omitted here for the sake of space (for example optionality or nullability) but could be easily included.

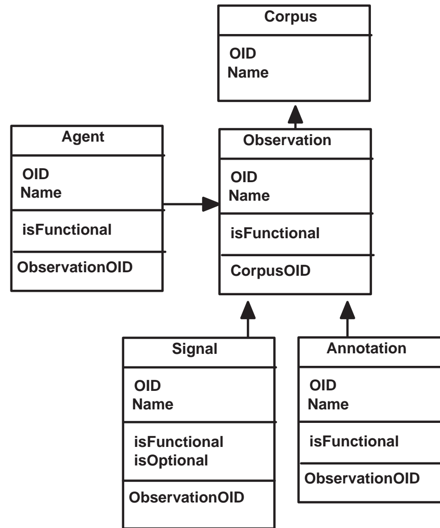


Figure 4.5: The NXT Metamodel.

Let us conclude the section by listing and commenting the metaconstructs of interest, with the correspondences in the two models (it should be noted that the correspondence is in some cases not really precise, but this could be dealt with by considering "variants" of constructs, which are omitted in this preliminary discussion):

SM-ABSTRACT. (we use the prefix sm for this and the other constructs, as they belong to the supermodel) The construct SM-ABSTRACT is used to describe all the kinds of objects, concrete or abstract. It corresponds to constructs used in many models, such as ER model entities, semantic network nodes, NOMOS things and NXT corpora. Each object must have an identity to be uniquely identifiable. There are variants for this construct, corresponding to classes, instances, and properties, distinguished by means of attributes.

SM-COLLECTION. It models collections of objects of the same type, with variants corresponding to the specific form (such as set, list, or bag). This is not really essential here, but could be needed for other platforms, and we have included it here for the sake of completeness.

SM-OBJECT. It is the generalization of SM-ABSTRACTand SM-COLLECTIOn, described above.

SM-AGGREGATIONOFABSTRACT. It allows the definition of a relationship on two or more SM-ABSTRACT components and corresponds to relations in NOMOS (as well as relationships in the ER model).

SM-COMPONENTOFAGGREGATION. This construct allows the specification of the participation of an object to an SM-AGGREGATIONOFABSTRACT, this includes the definition of the role in NOMOS relation (i.e. Subject or Object thing).

SM-PARTICIPANTOFABSTRACT. It allows the definition of the actors of an SM-ABSTRACT. This models the Entity concept in NOMOS and the Agent one in NXT.

SM-SIGNAL. This construct corresponds to the constructs used in many models of semantic annotations to represent the different multimedia sources that characterize the interaction that must be annotated.

SM-ATTRIBUTEOFSIGNAL. It describes a property of a signal. It can even be represented by a literal, and also define signals features.

SM-ANNOTATION. This construct models the concept of the annotation. It can be either an abstract or even a literal.

SM-TYPE. An assertion is related to a class that expresses its meaning.

SM-IDENTITY. It describes the correspondence between an SM-OBJECT and a form of identification for it.

Figures 4.6 and 4.7 show the correspondence between the constructs in the two models of interest and those in the supermodel.

For the purposes of semantic annotations interoperability, translations from a model to another can be carried out with the strategy defined in Chapter3.3, namely:

a) import of the source data to the supermodel;

b) translation within the supermodel;

c) export of the data from the supermodel to the target model.

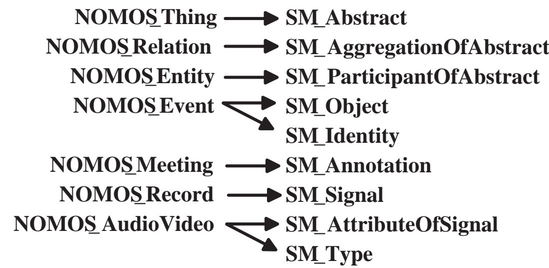


Figure 4.6: Correspondence between NOMOS and the supermodel.

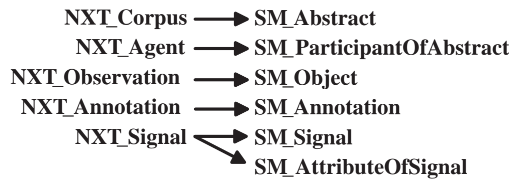


Figure 4.7: Correspondence between NXT and the supermodel.

By definition of supermodel, the first and the third steps are essentially copy operations and so they can be automatically managed once the correspondence between the constructs of the models and the constructs of the supermodel is built. In Fig. 4.6 the correspondences represented by a single arrow can be implemented by a plain copy operation from the metamodel to the supermodel tables. In the cases the figure shows a double arrow, the copy is slightly more complex: an element of NOMOS_Event has a correspondence with an element in SM-OBJECT and an element in SM-IDENTITY: the identifier is represented in SM-OBJECT whereas in SM-IDENTITY there are information on the identifying mechanism.

Translations within the supermodel can then be carried out by eliminating constructs that appear in the current set of data and are not allowed in the target model.

The translations we are considering here can be implemented by noting

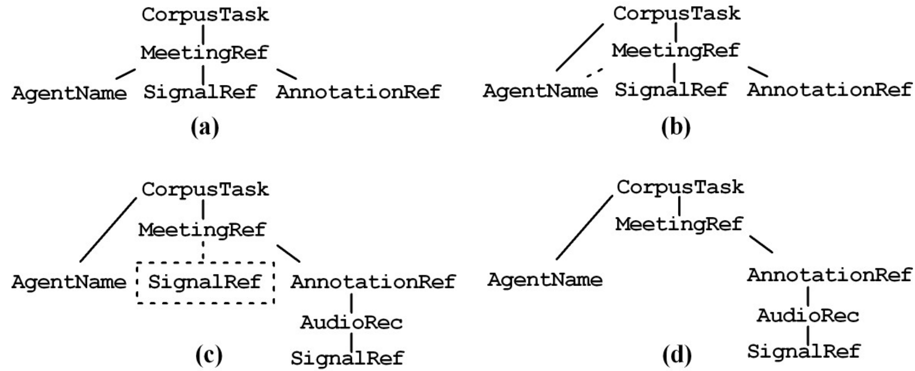


Figure 4.8: A simple translation from NXT to NOMOS.

which are the constructs available in one model and not available in the other. Considering a translation from NOMOS to NXT we will have to eliminate SM-AGGREGATIONOFABSTRACT by replacing them with other constructs, for example new SM-ABSTRACTs (see Figures 4.6 and 4.7). We can roughly state that this is a translation similar to the standard translation in conceptual models that replaces n-ary relationships by means of binary ones. A complete translation can then be built by specifying the basic translations needed to replace the constructs that are not available in the target model and then composing them.

In Fig. 4.8 a simple example of translation is sketched.

Starting from an NXT schema model (Fig. 4.8(a)) the system performs a few basic translations in order to obtain a model compliant to NOMOS (Fig. 4.8(d)). The first step is the change of the connection of the `AgentName` as shown in Fig. 4.8(b), in other words we must change the reference to the original containing object. Fig. 4.8(c) shows intuitive sketches of the intermediate schemas in this process. The translation into the NOMOS environment requires the introduction of a new object (`AudioRec`) for the signal class of NXT model associated to the annotation (`AnnotationRef`) and the `SignalRef` becomes a child of this one. As we said before, translations are performed by composing elementary steps translating NXT metaconstructs into NOMOS metaconstructs via the supermodel constructs. The last step produces a schema in the NOMOS model as shown in Fig. 4.8(d).

4.3 RDF vs Topic Maps

To test the validity of our approach we have also considered two Semantic Web languages for whom interoperability issues have been often addressed by the research community, namely RDF and Topic Maps [Gar05] .

The Resource Description Framework (RDF)[1] is a model that was born as the base model for the Semantic Web. It is used for representing information about resources in the World Wide Web. It is particularly intended for representing metadata about Web resources. RDF aim is that of providing a tool to describe formally the information the Web contains to make it machine-processable.

The key element of such a model is the *resource*. Together with the concept of resource two other simple elements contribute to the RDF model: *properties* and *statements*. A resource can be a Web page, an entire Web site, an element within a document, or an abstract concept. It is identified uniquely by a URIref, i.e. a URI [BLFM05] plus an optional anchor. A property is associated to a resource and allows its description by means of a specific characteristic. A statement is a triple formed by a subject, a predicate and an object in the form of a triple $< subject, predicate, object >$, where the subject is always a resource, the predicate is a property and the object is the value of the property. Furthermore RDF defines a mechanism allowing the reification of statements, that is a way to assert facts about a statement. To reify a statement means to make a statement being the subject and object of other statements. Due to the statement structure, an RDF document can then be seen as a directed labeled graph.

The focus of Topic Maps [GM05] is conceptual organization of information, primarily with a view to "findability", but the technology has much wider applicability. A major difference with RDF is the greater focus on human semantics in Topic Maps, as opposed to the "machine semantics" (or automated reasoning) of RDF. Topic Maps consist of topics, each of which represents some thing of interest, known as subjects (the definition of subject is deliberately all-inclusive). Associations represent relationships between two or more topics, they have a type (which is a topic) and consist of a set of (role type, role player) pairs (where both elements are topics). Occurrences can be either simple property-value assignments for topics, or references to resources which contain information pertinent to a topic. Occurrences also have a type (which is a topic). In addition, topics may have one or more names (which also may

[1] Resource Description Framework (RDF). `http://www.w3.org/RDF`

be typed).

As in RDF topics can have attached identifying URIs. The difference is that each URI can be either a subject locator (which means that dereferencing it produces the information resource the topic represents) or a subject identifier (which when dereferenced produces an information resource describing the subject). Another difference is that a topic can have any number of identifying URIs of either type attached. There is a defined procedure for merging topic maps based on these identifiers. Further, any construct can be reified (associations, occurrences, roles, and names), and any construct (except roles) may have a scope, which is a set of topics representing the context in which the construct is valid.

To enable the supermodel representing RDF and Topic Maps, reflecting the above mentioned peculiarities, we have enriched some of the existing metaconstructs with Semantic Web specific properties and other new metaconstructs are introduced.

Devising a generic mapping can sound as the ideal way to perform translations, avoiding to force a model to behave as the other, implementing translations in a more natural way. But when we create the mappings between RDF and Topic Maps, it arises the problem of a lack of information because of the different expressivity of the two models.

Let us translate the RDF statement `<S,P,O>` in Topic Maps. It is an assertion that can be translated in an *Association*, an *Occurrence* or a *Name* and it cannot be known, a priori, which is the right choice. We need to evaluate the meaning of the property P to select which is the right target construct. The result is a vocabulary-specific mapping that nevertheless allows us to reach satisfiable results in terms of keeping the expressiveness of the model and its meaning during the translation process.

As we will demonstrate with the following example, the use of Datalog rules allows us to naturally implement the vocabulary-specific mappings by means of conditional expressions that select the target construct. Moreover rules are written at the metalevel between the representations of source and target model in terms of metaconstructs, keeping the independence from a given model and allowing MIDST to involve new models reusing the previously made work.

The meaning of SM-ABSTRACT is extended to represent symbols that are used to describe the knowledge of interest in RDF and Topic Maps. With this metaconstruct we describe both the nodes of an RDF graph and topics of a Topic Map. To specify the node (topic) properties, as if it would be a resource with an URI or a literal, we use the SM-LEXICAL metaconstruct. This allows us to define a generic transformation between the structure of a graph and a

map keeping the particular resources separated from the concept of resource. SM-LEXICALs are also used to express the different kinds of identification that Topic Maps allow, namely SubjectIndicator and SubjectAddress. Examining the way RDF and Topic Maps relate concepts, we have noticed that they can basically fall into already defined metaconstructs with some extensions. For example, SM-BINARYAGGREGATIONOFABSTRACTS, can be used to describe the Topic Maps Occurrence and Name relationships extending it with an optional reference to an SM-ABSTRACT that represents the scope that we generically call Qualification. The same construct can describe the RDF Statement, for this purpose we have introduced the isDirected property that allows the representation of RDF directed arcs.

The SM-AGGREGATIONOFABSTRACTS metaconstruct has been extended with two optional references to SM-ABSTRACT that represent the type and the qualification of the aggregation, this is used for the Association of Topic Maps. The type and qualification are expressed as abstracts since they are topics. Each SM-COMPONENTOFAGGREGATIONOFABSTRACT of the enhanced SUPERMODEL has a reference to an abstract that is the role of the participation, a topic itself. The introduction of the new metaconstruct TYPE, which expresses the typing relation between SM-ABSTRACTs, allows us to model the RDF construct rdf:type and the equivalent for Topic Maps that is instanceOf.

We have provided the supermodel with these new constructs through which we can now illustrate how to perform a translation between Topic Maps and RDF. Figures 4.9 and 4.10 shows the supermodel portion of interest respectively for RDF and TopicMaps (each of them with their metaconstructs and properties).

Let us consider the following simple RDF document that illustrates the knowledge between two persons, one of which with an homepage. The example uses the FOAF vocabulary [2].

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:foaf="http://xmlns.com/foaf/0.1/">
    <foaf:Person>
      <foaf:name>John Smith</foaf:name>
      <foaf:homepage rdf:resource="http://www.john.sm"/>
      <foaf:knows>
        <foaf:Person>
```

[2] The Friend of a Friend (FOAF) project, http://www.foaf-project.org/.

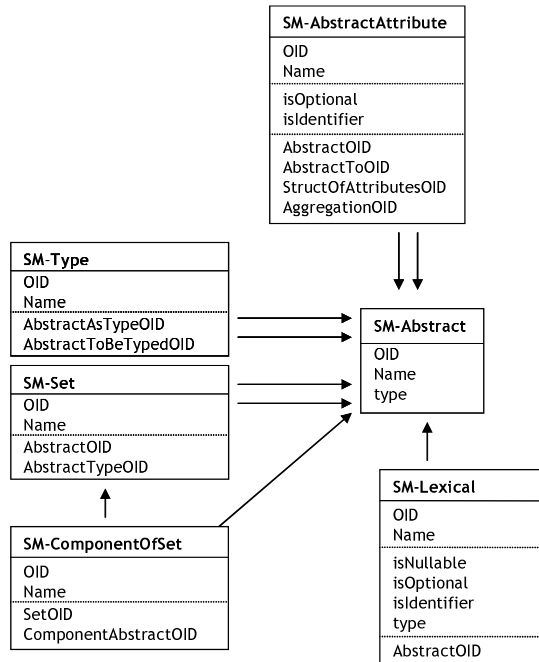


Figure 4.9: A portion of the supermodel showing the RDF-related meta-costructs.

```
          <foaf:name>Frank Red</foaf:name>
        </foaf:Person>
      </foaf:knows>
    </foaf:Person>
</rdf:RDF>
```

The related graph in sketched in Figure 4.11:

When translating the RDF `Statement` into Topic Maps, the Datalog rules generate a `Name` for the property `foaf:name`, an `Occurrence` for the property `foaf:homepage` and an `Association` for the property `foaf:knows`. In the latter situation, we don't have the information of the association roles in RDF while Topic Maps miss the direction of the statement. We choose to assign the roles values of `rdf:subject` and `rdf:object` to the association participants, in order to keep the semantic of the direction of the RDF arcs. As

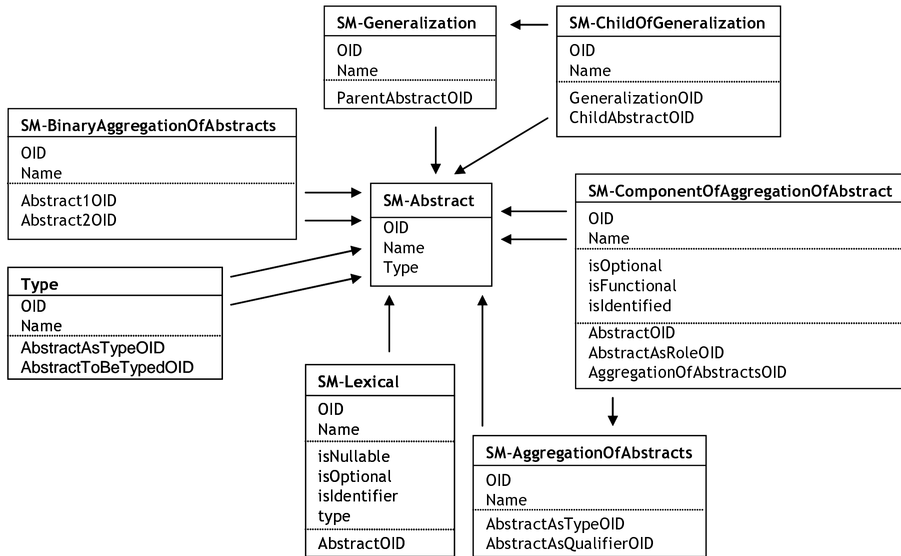


Figure 4.10: A portion of the supermodel showing the TM-related meta-costructs.

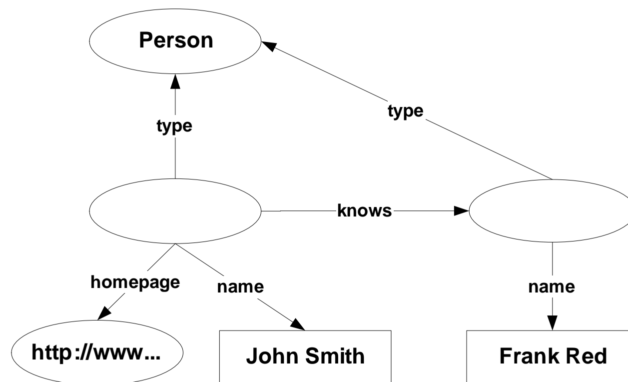


Figure 4.11: A simple RDF graph.

result of the translation we obtain the following Topic Maps (we use the XTM syntax [Top01]):

```
<topic id="id78">
 <instanceOf>
  <subjectIndicatorRef xlink:href="foaf:Person"/>
 </instanceOf>
 <baseName>
  <baseNameString>John Smith</baseNameString>
 </baseName>
 <occurrence>
  <instanceOf>
   <subjectIndicatorRef xlink:href="foaf:homepage"/>
  </instanceOf>
  <resourceRef xlink:href="http://www.john.sm"/>
 </occurrence>
</topic> <topic id="id32">
 <instanceOf>
  <subjectIndicatorRef xlink:href="foaf:Person"/>
 </instanceOf>
 <baseName>
  <baseNameString>Frank Red</baseNameString>
 </baseName>
</topic> <association>
 <instanceOf>
  <subjectIndicatorRef xlink:href="foaf:Knows"/>
 </instanceOf>
 <member>
  <roleSpec><subjectIndicatorRef xlink:href="rdf:subject"/></roleSpec>
  <topicRef xlink:href="#id18"/>
 </member>
 <member>
  <roleSpec><subjectIndicatorRef xlink:href="rdf:object"/></roleSpec>
  <topicRef xlink:href="#id32"/>
 </member>
</association>
```

On the other side, considering the translation from Topic Maps to RDF, specifically the case of translating an association, we have to translate n-ary relationships to binary statements. In Fig. 4.12 there is a topic map that represents the employment association between the company HiTech, with the role of employer and two employees.

To translate the Topic Map of Figure 4.12 to RDF, we define a rule that creates an SM-ABSTRACT for each topic, with SM-LEXICALs that represent the identification and the names. Other simple rules are devoted to the creation of an SM-ABSTRACT without SM-LEXICALs (blank node) for each participant, that allows us to define an SM-BINARYAGGREGATIONOFABSTRACTS that links the participant topic with the role topic. We then create an SM-ABSTRACT for the association and an SM-ABSTRACT for the type that are linked by an SM-TYPE metaconstruct.

Finally, members are linked to the association through the use of an SM-BINARYAGGREGATIONOFABSTRACTS. Exploiting the correspondence between

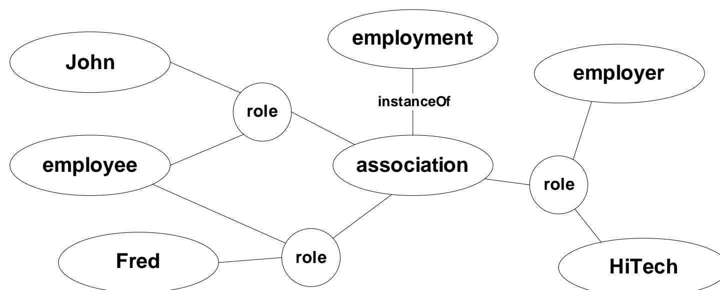


Figure 4.12: Topic Maps example.

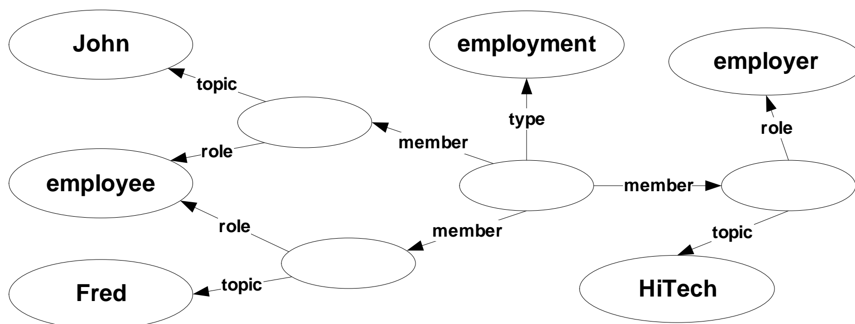


Figure 4.13: Resulting RDF graph.

the supermodel and RDF we can eventually generate the graph as illustrated in Figure 4.13.

4.4 Discussion

We have shown how ModelGen approach can be at the basis of a framework for the translation of annotations from a platform to another. In this chapter two semantic annotation platforms, NOMOS and NXT have been addressed. However, our approach can be extended to similar platforms which have a schema-based structure.

We have also illustrated a typical scenarios in which Semantic Web formalisms (namely RDF and Topic Maps) can be described by our Supermodel

and how our framework can be used to define translations with a high-level, model independent approach.

Chapter 5

Bridging the Gap between Semantic Annotations and Databases

In the previous chapter we analyze two interesting examples of application of our semantic annotation interoperability approach, one at system level the other at language level. The focus of the next three chapters is to enhance the aforementioned approach in order to manage interoperability between (relational) databases and semantic annotations described as OWL ontologies. In this chapter we present the main reasons of this work and the most recent researches on this topic.

5.1 Motivation

Starting from the analysis of the available approaches that propose to face the problem of the translations of schemes and data between ontologies and databases, it is possible to underline several aspects that motivate our research.

From Database to Ontology

First of all we can consider the problem of integration of heterogeneous sources of information. Ontologies play a key role in this process. The term "ontol-

ogy" is borrowed from philosophy, where *Ontology* is a systematic account of *Existence*. In computer science an ontology is formally a defined system of concepts. An ontology is "a formal, explicit specification of a shared conceptualization" [Gru95]. Conceptualization corresponds to an abstract model of a domain which identifies the relevant concepts and their relationships. Explicitly it means that the used concepts are unique and their usage is formally confined. Formal refers to the fact that ontologies should be machine-readable. Shared indicates that an ontology is accepted by a group of people and used cooperatively.

Ontologies can be used to define semantic mapping between different information sources. A typical example is the integration of data coming from old legacy systems. Autonomous database systems usually have incompatible schemas making interoperability among them difficult. For a long time, this has been recognized as a schema mapping and data integration problem [DH05] [Len02] [SE05]. In simplest terms, database integration requires (i) mapping systems that define the relationships (mappings) among database schemas and (ii) integration systems that use those mappings to answer queries or translate data across database sources. In addition to the more expressive representations offered by ontologies, they allow integration to cover a larger variety of structured data in theory, although it raises the question of adequate performance in real systems.

Another important aspect to consider is the acquiring of domain ontology that still requires great efforts. Therefore, it is necessary to develop methods and techniques that allow reducing the effort necessary for the knowledge acquisition process. Database schemata, in particular the conceptual schemata modeled in semantic data models such as the Entity-Relationship (ER) model contains (implicitly) abundant domain knowledge. Extracting the knowledge from them can thus profitably support the development of Web ontologies.

Also the so called *Deep Web* arises similar problems. Nowadays, indeed, a large percentage of Web pages are not static documents. On the contrary, the majority of Web pages are dynamic. Therefore the majority of information on the Web is generated from underlying databases (i.e. the deep Web). To discover content on the Web, search engines use Web crawlers that follow hyperlinks. This technique is ideal for discovering resources on the surface Web but is often ineffective at finding deep Web resources. Moreover, it is not possible to semantically annotate the dynamic pages generated from the data sources. Existing tools can only produce semantic annotations for static Web pages and how to annotate dynamic Web pages that are generated from the underlying databases (the greater majority of current Web content [CHL+04]) when the

clients request the pages is still an open problem[RH05] [BC02]. This problem has been referred to as *deep annotation* [VHS+04] [HSV03] that means the process of creating ontological instances for the database-based, dynamic contents by reaching out to the Deep Web and directly annotating the underlying database of the dynamic Web site.

One of the research fields which has recently gained much scientific interest within the database community are Peer-to-Peer databases, where peers have the autonomy to decide whether to join or to leave an information sharing environment at any time. The principle is: data stored on one single peer has to be made accessible to other remote peers and vice versa. Afterwards this data can be requested, queried, replicated, or integrated depending on the purpose of the remote system. As a result, sharing relational data within a Peer- to-Peer environment means to distribute not only data items themselves, but also their schemas among multiple previously unknown peers. We thus need an exchange format, which on the one side can be understood by a broad community of peers without being explicitly arranged beforehand and which on the other side has to be suitable for representing relational schemas and their corresponding data instances. This can be suitably achieved through the use of ontologies without having to define a schema and data exchange format explicitly.

From Ontology To Database

The ontologies are important for the integrations of data and of applications, besides they facilitate the communication between human beings and informative systems. However to benefit from the large amount of information stored in ontologies it is necessary to have efficient and effective ways to manage and query those data.

Large ontologies are often stored in database repositories in order to exploit their ability to handle secondary storage and to answer queries in efficient ways. Actually, several tools for managing (building, inferring, querying, etc.) ontology data and ontology-based data are available (e.g. Protégé [NFM00] [NSD+01]). Usually, ontology-based data manipulated by these tools are stored in the main memory. Thus, for applications manipulating a large amount of ontology-based data, query performance becomes a new issue. Therefore, due to the maturity of database systems, it is possible to efficiently store large ontologies (with million of instances) getting benefit from the functionalities offered by DBMSs (i.e. query performance, efficient storage, transaction management, etc.).

5.2 Related Work

As stated in the previous section we are interested in semantic annotations that have an high level of formality, i.e. those annotations that can be represented by means of ontologies.

Various proposals exist that describe the translation between ontologies and databases, but the majority of this approaches study only one-way transformations (i.e. ontology to database or database to ontology).

We can classify the different approaches in two main branch:

- dictionary-based
- dictionary-independent

In the first kind of approaches a sort of dictionary is used to define the guideline of the transformations between ontology and database. To represent a relational schema and its instances, ad-hoc ontologies are created as in [dLC05] and [TBA06]. Both approaches describe the database schema and instances by means of a suitable dictionary used in an ontology.

In [TBA06], the authors propose an approach that helps the domain experts to quickly generate and publish OWL ontologies describing the underlying relational database systems while preserving their structural constraints. The generated ontologies are constructed using a set of vocabularies and structures defined in schema that describes relational database systems on the Web so they guarantees that user applications can work with data instances that conformed to a set of known vocabularies and structures.

Handschuh et al. [HSV03] apply a similar approach for annotating data intensive Web sites. In these works, mappings are managed referring to specific ontologies that describe the source relational model. Similar approaches are followed by [Kri06] and [Lau08] for the generation of RDF documents. Das et al. [DCES04] proposed a solution to extracting data from the OWL document, and then storing data in relational database. It uses a *reference ontology* as a dictionary. It also enables users to reference ontology data directly from SQL using the semantic match operators.

Dictionary-independent approaches do not use a prefixed dictionary to perform the translations. This include approaches that use Description Logics or machine learning techniques such as [Hab07]. These kind of approaches do not use ad hoc ontologies but aim at representing the semantics of the information in the relational database source.

Cullot et al. [CGY07] and Shen et al. [SHZZ06] are representative examples.

In [CGY07] the DB2OWL tool is described. It looks for some particular cases of database tables to determine which ontology component has to be created from which database component. The created ontology is expressed in OWL-DL language [1] which is based on Description Logics.

Shen et al. [SHZZ06] propose the rules of mapping relational model to OWL for the data integration, and they are classified as concepts, properties, restrictions and instances. These rules can be applied to mapping relational database to ontologies in OWL, whereby the mapping and transferring can be performed (semi-)automatically. The rules for concepts, properties and restrictions depict the correspondence at metadata level, which avoid migrating the large amount of data. The rules for instances are applied to create data for exchanging at running time. All the rules can also be applied to learning ontologies from relational database.

A number of formal approaches based on description logics exist (see for example [XCDS04] and [MHS07]). These cannot be compared with our work, as we concentrate on structural aspects and so we do not refer to reasoning capabilities. Moreover, these pieces of work refer to one specific data model, ER or relational, whereas our approach applies to many different data models, belonging to many families, including relational, ER, object-oriented and object-relational.

While the dictionary-based approaches are preferable in case of ontologies creation as an interchange format, the dictionary-independent approaches find their greatest in sharing knowledge, extrapolated from relational databases, on a particular domain of interest.

5.3 Discussion

In this chapter we have introduced the motivations of translating between databases and ontologies. Moreover we have discussed some of the most relevant researches that we have used as a starting point for the definition and extension of our MIDST project.

Comparing to these works, our approach aims at a greater generality. We describe the models of interest by a metamodel, subsequently schema and instances are treated. This allows us to be model independent, with a general approach that is extensible to virtually any model. Moreover, translations are not embedded but specified by means of high level rules and therefore they are customizable according to the different needs.

[1] http://www.w3.org/TR/owl-features/

Moreover, the majority of the aforementioned approaches considers translations in both directions, nor any form of model independence, as each of them is tightly related to a specific data model and a specific translation approach.

Chapter 6

OWL and Relational Database Mappings

Interoperability of ontologies and databases has received a lot of attention recently. However, most of the work has concentrated on specific problems (such as storing an ontology in a database or making database data available to ontologies) and referred to specific models for each ones. Here we describe how our approach can be exploited to manage also this kind of transformations in both directions (ontologies to databases and vice versa).

As we observed in Chapter 3.3, the starting point of our approach is the idea that a *metamodel* is a set of constructs (called *metaconstructs*) that can be used to define models, which are instances of the metamodel. Therefore, we actually define a model as a set of constructs, each of which corresponds to a metaconstruct. An even more important notion, is the *Supermodel*: it is a model that has a construct for each metaconstruct, in the most general version. Therefore, each model can be seen as a specialization of the supermodel, except for renaming of constructs.

6.1 Towards Ontology and Databases Integration

In order to achieve the complex tasks of interoperability and integration of databases and ontologies it is necessary to firstly define each model in term of supermodel metaconstructs. In the following subsections we show our representations of (relational) database and ontology data models by means of metaconstructs. The first one is already present in the previous version of the MIDST project, the second is completely new and it belongs to the work of extending our approach to Semantic Web and in particular to Semantic Annotation defined by means of ontologies.

Relational Data Model

The relational data model is the standard model for logical design of databases. The most important construct is the relation, which consists of a heading and a body. A heading is a set of attributes, while a body (of an n-ary relation) is a set of n-tuples. Each relation is a table, therefore we consider a relational model with tables constituted by columns of a specified type; each column could allow `null` value or be part of the primary key of the table. Moreover we can specify foreign keys between tables involving one or more columns.

The Figure 6.1 shows the constructs of the relational metamodel in a UML-like class diagram.

Following the MIDST conventions each construct is made of four parts:

- the construct name, that is unique in the model;
- a list of required attributes. Generally we have:

  OID that is the unambiguous identifier of each instance of the construct

  Name the name that identifies the instance of the construct

- a list of properties that defines the characteristics of the construct
- a list of references to other constructs that establishes the relationships between the different constructs in the model

In the following description, for each construct we show the correspondent representation in terms of supermodel metaconstruct.

Table SM-AGGREGATION. Each table in the relational model can be seen as a set (or "aggregation") of columns. Therefore we map tables with aggregations of lexicals.

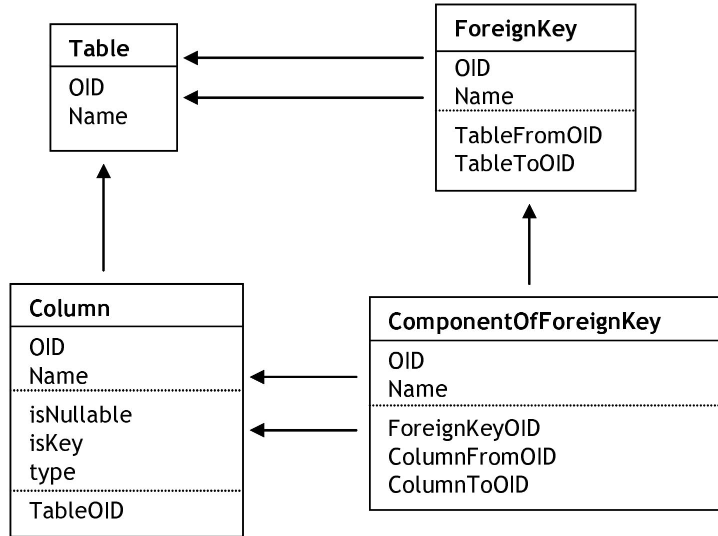


Figure 6.1: Relational Metamodel.

Column SM-LEXICAL. We can specify the data type of the column (`type`) and whether it is part of the primary key (`isIdentifier`) or it allows null value (`isNullable`). It has a reference toward an SM-AGGREGATION.

Foreign Key SM-FOREIGNKEY and SM-COMPONENTOFFOREIGNKEY. With the first construct (referencing two SM-AGGREGATIONS) we specify the existence of a foreign key between two tables; with the second construct (referencing one SM-FOREIGNKEY and two SM-LEXICALS) we specify the columns involved in a foreign key.

The relational metamodel is shown in Figure 6.2.

OWL Synopsis

Before describing the OWL data model it is necessary to describe the characteristics of the language in order to better understand our choices.

The Web Ontology Language (OWL) was designed to add the constructs of

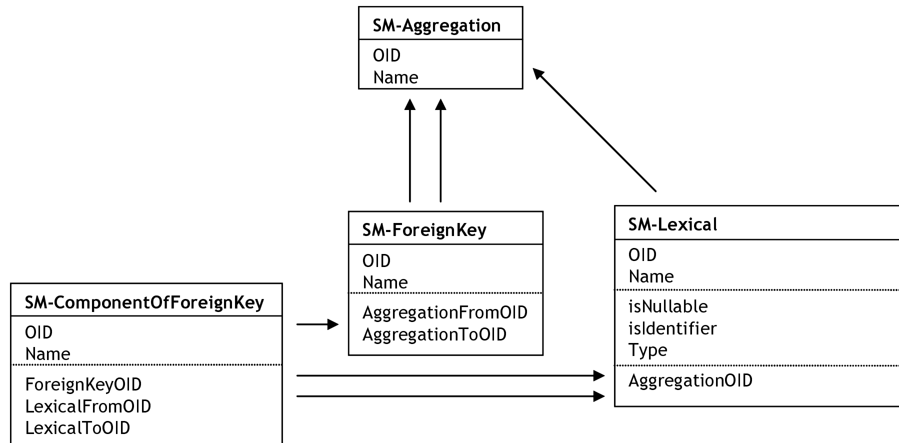


Figure 6.2: Relational Metamodel in terms of Supermodel metaconstructs.

Description Logics (DL) to RDF[1], significantly extending the expressiveness of RDF Schema both in characterizing classes and properties. Description Logics are a set of knowledge representation languages with formal semantics based on their mapping to First Order Logic (FOL). Description Logics have been extensively studied since the 1980s, including studies on the tradeoffs between the expressivity of the chosen language and the efficiency of reasoning.

OWL has been designed in a way that it maps to a well-known Description Logic with tractable reasoning algorithms. The Web Ontology Language is in fact a set of three languages with increasing expressiveness: OWL Lite, OWL DL and OWL Full. These languages are extensions of each other ($OWL_{Lite} \subseteq OWL_{DL} \subseteq OWL_{Full}$) both syntactically and semantically. For example, every OWL Lite document is a valid OWL DL document and has the same semantics when considered as an OWL DL document, e.g. it leads to the same logical conclusions. The vocabularies of these languages extend each other and languages further up in the hierarchy only relax the constraints on the use of the vocabulary.

Although it is generally believed that languages of the OWL family would be an extension of RDF(S)[2] in the same sense, this is only true for OWL Full,

[1] Resource Description Framework (RDF), `http://www.w3.org/RDF/`
[2] RDF Schema, `http://www.w3.org/TR/rdf-schema/`.

the most expressive of the family ($RDF(S) \subseteq OWL_{Full}$).

The middle language, OWL DL was the original target of standardization and it is a direct mapping to an expressive Description Logic. This has the advantage that OWL DL documents can be directly consumed by most DL reasoners to perform inference and consistency checking. The constructs of OWL DL are also familiar, although some of the semantics can be surprising mostly due to the open world assumption [RDH+04]. Description Logics do not allow much of the representation flexibility introduced above (e.g. treating classes as instances or defining classes of properties) and therefore not all RDF documents are valid OWL DL documents and even the usage of OWL terms is limited. For example, in OWL DL it is not allowed to extend constructs of the language, i.e. the concepts in the RDF, RDF Schema and OWL namespaces.

In the case of the notion of a Class, OWL also introduces a separate owl:Class concept as a subclass of `rdfs:Class` in order to clearly distinguish its more limited notion of a class. Similarly, OWL introduces the disjoint classes of object properties and datatype properties. The first refers to properties that take resources as values (such as `foaf:knows`) and the latter is for properties ranging on literals such as `foaf:name`. OWL Full is a limitless OWL DL: every RDF ontology is also a valid OWL Full ontology and has the same semantics when considered as an OWL Full document. However, OWL Full is undecidable, which means that in the worst case OWL Full reasoners will run infinitely. OWL Lite is a lightweight sub-language of OWL DL, which maps to a less expressive but even more efficient DL language. OWL Lite has the same limitations on the use of RDF as OWL DL and does not contain some of the terms of OWL DL. In summary, RDF documents are not necessarily valid OWL Lite or OWL DL ontologies despite the common conviction (see also the classical semantic Web layer cake in Figure 6.3). In fact, downgrading a typical RDF or OWL Full ontology to OWL DL is a tedious engineering task. It typically includes many simple steps such as declaring whether properties are object properties or datatype properties and importing the external ontologies used in the document, which is mandatory in OWL but not in RDF.

However, the process often involves more fundamental modeling decisions when it comes to finding alternate representations.21 Most existing Web ontologies make little use of OWL due to their limited needs, but also because general rule-based knowledge cannot be expressed in OWL. The additional expressivity of OWL, however, is required for modeling complex domains such as medicine or engineering, especially in supporting classification tasks where we need to determine the place of a class in the class hierarchy based on its description.

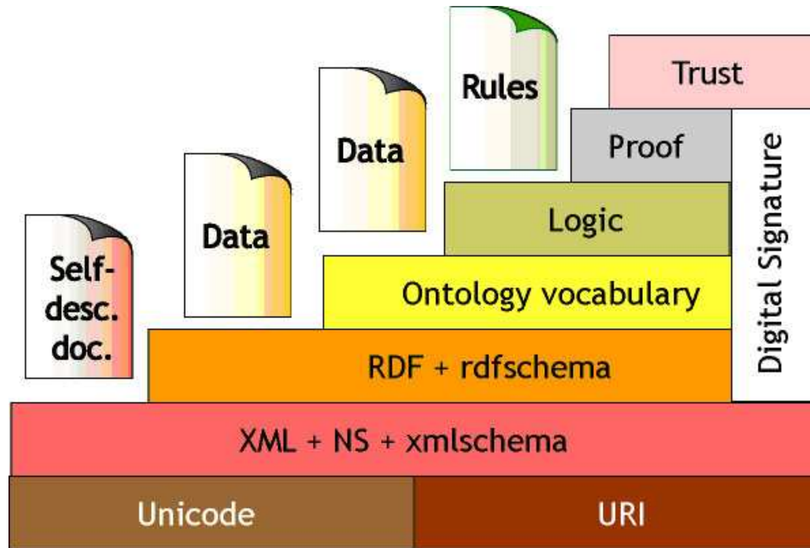


Figure 6.3: The Semantic Web layer cake.

In our experiments we consider the OWL Lite model, that is sufficient to demonstrate the features of our approach and that can be simply encoded in RDF. In the following, we always refer to OWL Lite also if some discussion can be extended to OWL DL.

In Figure 6.4 the main constructs of OWL Lite are listed.

The most important construct of OWL is the *Class*. Classes provide an abstraction mechanism for grouping resources with similar characteristics. Each class is generally associated with a group of *individuals*. These individuals are instances of the class. Classes can be organized in a specialization hierarchy using `rdfs:subClassOf`. There is a built-in most general class named `Thing` that is the class of all individuals and is a superclass of all OWL classes. On the contrary, `Nothing` is the class that has no instances and a subclass of all OWL classes.

Properties can be used both to state relationships between individuals (`ObjectProperty`) and from individuals to data values (`DatatypeProperty`). Each property may have a *domain* and a *range*. A domain (`rdfs:domain`) of a property limits the individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a

RDF Schema Features:

- *Class (Thing, Nothing)*
- *rdfs:subClassOf*
- *rdf:Property*
- *rdfs:subPropertyOf*
- *rdfs:domain*
- *rdfs:range*
- *Individual*

(In)Equality:

- *equivalentClass*
- *equivalentProperty*
- *sameAs*
- *differentFrom*
- *AllDifferent*
- *distinctMembers*

Class Intersection:

- *intersectionOf*

Property Characteristics:

- *ObjectProperty*
- *DatatypeProperty*
- *inverseOf*
- *TransitiveProperty*
- *SymmetricProperty*
- *FunctionalProperty*
- *InverseFunctionalProperty*

Property Restrictions:

- *Restriction*
- *onProperty*
- *allValuesFrom*
- *someValuesFrom*

Restricted Cardinality:

- minCardinality *(only 0 or 1)*
- *maxCardinality (only 0 or 1)*
- *cardinality (only 0 or 1)*

Figure 6.4: The main constructs of OWL Lite.

class as one of its domains, then the individual must belong to the class. The range (`rdfs:range`) of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. Domain and range are called global restrictions since the restriction is stated on the property and not just on the property when it is associated with a particular class.

OWL Lite allows (local) restrictions to be placed on how properties can be used by instances of a class. There are two main kinds of restriction, namely *allValuesFrom* and *someValuesFrom*. These restrictions (and the cardinality restrictions) are used within the context of an `owl:Restriction`. The `owl:onProperty` element indicates the restricted property. The restriction *allValuesFrom* is stated on a property with respect to a class. It means that this property on this particular class has a local range restriction associated with it. Thus if an instance of the class is related by the property to a second individual, then the second individual can be inferred to be an instance of the local range restriction class. The restriction *someValuesFrom* is stated on a property with respect to a class. A particular class may have a restriction on a property that at least one value for that property should be of a certain type.

There are special identifiers in OWL Lite that are used to provide information concerning properties and their values. Besides inverse, symmetric

and transitive properties that are clear, we ave also *functional* and *inverse functional* properties. Properties may be stated to have a unique value. If a property is a *FunctionalProperty*, then it has no more than one value for each individual (it may have no values for an individual). This characteristic has been referred to as having a unique property. *FunctionalProperty* is shorthand for stating that the property's minimum cardinality is zero and its maximum cardinality is 1. Properties may be stated to be inverse functional. If a property is inverse functional then the inverse of the property is functional. Thus the inverse of the property has at most one value for each individual.

OWL can also represent equality or inequality features. Despite to describe all (in-)equality constructs we only want to focus on *sameAs*, *differentFrom* and *AllDifferent* constructs. With the *sameAs* construct two individuals may be stated to be the same. These constructs may be used to create a number of different names that refer to the same individual. With *differentFrom* an individual may be stated to be different from other individuals. Finally, a number of individuals may be stated to be mutually distinct in one *AllDifferent* statement.

OWL Lite allows intersections of named classes and restrictions by means of the *intersectionOf* constructor. A typical example of intersection is sketched in the following (using RDF/XML syntax):

```
<owl:Class rdf:ID="WhiteWine">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Wine" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasColor" />
      <owl:hasValue rdf:resource="#White" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Classes constructed using the set operations are more like definitions than anything we have seen to date. The members of the class are completely specified by the set operation. The construction above states that `WhiteWine` is exactly the intersection of the class Wine and the set of things that are white in color. This means that if something is white and a wine, then it is an instance of `WhiteWine`. Without such a definition we can know that white wines are wines and white, but not vice-versa.

OWL Data Model

The most important construct of the OWL data model is Class that represents the concept of class. In Figure 6.5 the metamodel of OWL Lite is shown.

The construct Class represents classes both named or restricted: to know what type of class is identified by an instance of the construct, it is necessary to take into account the references to other constructs. For example, a class defined as a restriction on a property, will be referenced from the ClassAsRestrictionOID of RelationshipBetweenClasses construct, which represents relations between classes (both ObjectProperty and DatatypeProperty). A class defined as a finite intersection of other classes will be referenced from `ClassOID` of the Intersection construct. A class definition defined by identifier instead is simply represented by its name valued in the name of the Class construct.

The construct RelationshipBetweenClasses is used to represent an *objectProperty* and the equivalence between classes. In the case of an equivalence relation, the instance of this construct should have:

- the property `isEquivalence` set to `TRUE`;
- the property `isDirected` set to `FALSE` (because the direction of an equivalence relation is not important);
- references `class1OID` and `class2OID` refer to the two classes involved in the equivalence relationship;
- other properties and references are not considered.

In the case of an object property, the instance of this construct should have:

- the property `isSymmetric` is set to `TRUE` if the object property is an `owl:SymmetricProperty`;
- the property `isTransitive` is set to `TRUE` if the object property is an `owl:TransitiveProperty`;
- the property isDirected set to `TRUE` (because the direction of the property is established by the domain and range global restrictions);
- the property `isFunctional1` is set to `TRUE` if the object property is an `owl:FunctionalProperty` or the maximum cardinality is 1;

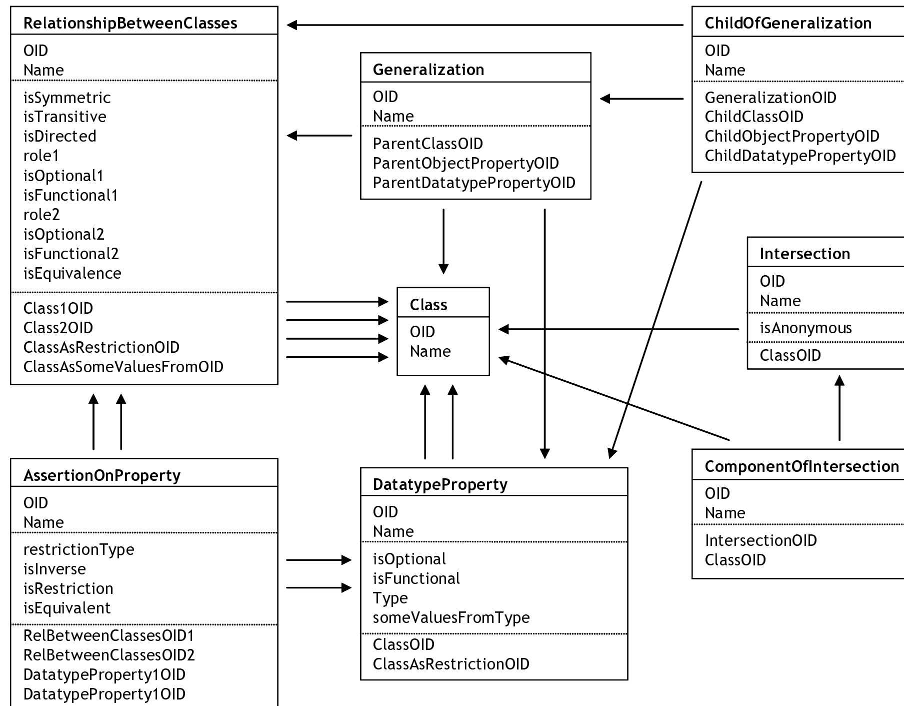


Figure 6.5: OWL Lite metamodel.

- the property `isFunctional2` is set to `TRUE` if the object property is an `owl:InverseFunctionalProperty`;
- references `class1OID` and `class2OID` refer respectively to the domain class and to the range class;

The different kinds of restriction modify the instance of the RELATIONSHIP-BETWEENCLASSES construct. The possibilities are summarized in Figure6.6.

Datatype properties are represented by the DATATYPEPROPERTY construct. Each instance of this construct should have:

- the property `isFunctional` is set to `TRUE` if the object property is an `owl:FunctionalProperty`;

Restriction Type	Attribute	Value
owl:AllValuesFrom	Class2OID	Refers to range class
owl:someValuesFrom	ClassAsSomeValuesFromOID	Refers to the class thast indicates the value of the OWL construct
owl:MinCardinality	isOptional1	FALSE if the minimum cardinality is 1; TRUE otherwise
owl:MaxCardinality	isFunctional1	TRUE if the minimum cardinality is 1; FALSE otherwise
owl:Cardinality	isOptional1, isFunctional1	The combinations of the two above situations

Figure 6.6: OWL Restriction for RelationshipBetweenClasses construct.

- the property `Type` contains the data type of the property (e.g. int, bool, etc.) ;
- reference `class1OID` refers to the domain class;

The construct AssertionOnProperty is used to represent the characteristics of a property (object property or datatype property). Let P_1 and P_2 two properties, with our construct it is possible to represent the following situations:

- P_1 is the inverse of P_2;
- P_1 is a restriction of P_2, with the type of restriction
- P_1 is equivalent to P_2

The AssertionOnProperty construct has a boolean attribute for each of the previous assertions, namely: `isInverse`, `isRestriction`, `isEquivalent`.

Through Intersection and ComponentOfIntersection constructs we model the intersection of two or more OWL classes.

Let be C a class that is the intersection of the classes C_1 and C_2, in order to represent this situation we need:

- to create an instance I of the construct Intersection in which `ClassOID` is related to the class C;
- to create an instance of the construct ComponentOfIntersection for each class that is a component of the intersection. In our case I_1 that is

related to C_1 and I_2 that is related to C_2. Both I_1 and I_2 must be related to the intersection I by means of the reference `IntersectionOID`.

The attribute `isAnonymous` in ComponentOfIntersection construct is set to `FALSE` value if the class defined as intersection is also defined by an URI.

The two OWL generalizations we consider (i.e. `rdfs:subClassOf` and `rdfs:subPropertyOf`), are represented through Generalization and ChildOfGeneralization. constructs. Generalization holds:

- the reference `ParentClassOID`, which referenced the Class construct, in the event of class generalization (`rdfs:subClassOf`);
- the reference `ParentObjectPropertyOID`, which referenced the RelationshipBetweenClasses construct, in the case of generalization of an object property (`rdfs:subPropertyOf`);
- the reference `ParentDatatypeProperty`, which referenced the DatatypeProperty construct, in the case of generalization of a datatype property (`rdfs:subPropertyOf`);

ChildOfGeneralization refers to the elements (children) of the generalization. For example, let C_1 a sub-class of class C_2, we have:

- an instance G of the Generalization construct in which the attribute `ParentClassOID` belongs to the class C_2;
- the instance C_G of the construct ChildOfGeneralization in which the attribute `ChildClassOID` belongs to the class C_1.

The table in Figure 6.7 shows the correspondences between the constructs of OWL Lite and our OWL data model constructs.

6.2 An Extended Supermodel

In the previous section we introduced both relational and owl metamodels. Relational metamodel is already included in the previous work, and the reader can find more details about it in [ACB06]. Here, we want to analyze the extension of the Supermodel in order to consider also ontologies (in particular OWL-compliant ontologies) and in general to address semantic annotation.

In order to show the generality of the approach we show how the supermodel can be modified and enhanced to represent those semantic Web elements. Due

Class Description			
OWL Lite	**OWL Model Construct**	**Attribute**	**Attribute Value**
owl:Class	Class		
owl:Restriction	Class RelationshipBetweenClasses AssertionOnProperty		
owl:allValuesFrom	RelationshipBetweenClasses	Class2OID	
owl:maxCardinality	RelationshipBetweenClasses	ClassAsSomeValuesFromOID	
owl:minCardinality	RelationshipBetweenClasses	isFunctional1	
owl:Cardinality	RelationshipBetweenClasses	isOptional1	
owl:intersectionOf	Intersection ComponentOfIntersection	isFunctional1 isOptional1	
Axioms			
OWL Lite	**OWL Model Construct**	**Attribute**	**Attribute Value**
rdfs:subClassOf	Generalization ChildOfGeneralization		
owl:equivalentClass	RelationshipBetweenClasses	isEquivalence	true
Properties			
OWL Lite	**OWL Model Construct**	**Attribute**	**Attribute Value**
owl:ObjectProperty	RelationshipBetweenClasses		
owl:DatatypeProperty	Lexical		
rdfs:subPropertyOf	Generalization ChildOfGeneralization		
rdfs:domain	RelationshipBetweenClasses Lexical	Class1OID ClassOID	
rdfs:domain	RelationshipBetweenClasses Lexical	Class2OID type	
owl:equivalentProperty	AssertionOnProperty	isEquivalent	true
owl:inverseOf	AssertionOnProperty	isInverse	true
owl:FunctionalProperty	RelationshipBetweenClasses Lexical	isFunctional1 isFunctional	true
owl:inverseFunctionalPrope rty	RelationshipBetweenClasses	isFunctional2	true
owl:SymmetricProperty	RelationshipBetweenClasses	isSymmetric	true
owl:TransitiveProperty	RelationshipBetweenClasses	isTransitive	true

Figure 6.7: Correspondences between OWL Lite elements and OWL data model constructs.

OWL Model	Supermodel
CLASS	SM-ABSTRACT
RELATIONSHIPBETWEENCLASSES	SM-BINARYAGGREGATIONOFABSTRACTS
DATATYPEPROPERTY	SM-LEXICAL
GENERALIZATION	SM-GENERALIZATION
CHILDOFGENERALIZATION	SM-CHILDOFGENERALIZATION

Figure 6.8: Correspondences between OWL Model and the Supermodel.

to the complexity of the whole Supermodel we consider only the constructs that are involved in the translation between semantic annotations and databases. In the following we briefly describe some of the major extensions made to the Supermodel.

The extendability of the Supermodel permits us to reuse some of the available constructs. Some of the most clear correspondences between the OWL model construct and the Supermodel metaconstructs are shown in Figure 6.8.

As we can see in the Figure 6.8 the SM-ABSTRACT can be used to represent the CLASS construct of OWL. Indeed, SM-ABSTRACT is used to represent abstract entities as it models, for example, the ENTITY construct of the Entity-Relationship model or the ROOTELEMENT construct of XML.

The SM-BINARYAGGRAGATIONOFABSTRACTS metaconstruct is used to represent the concept of binary relationship between two different abstract entities (SM-ABSTRACT). It includes the attributes `isOptional1`, `isOptional2`, `isFunctional1` and `isFunctional2`) that allow the definition of relationship cardinality in both sides. The attribute `isDirected` allows the definition of the relationship direction, while `Abstract1OID` and `Abstract2OID` belong to the SM-ABSTRACTs participating in the relation, whose roles are defined by `role1` and `role2` attributes.

The SM-LEXICAL metaconstruct represents the concept of lexical, i.e. a property with a primitive type value. The `isOptional` attribute allows to specify the minimum cardinality, the attribute `isNullable` specifies if it is allowed or not to have a `NULL` value. Obviously, `AbstractOID` belongs to the SM-ABSTRACT that owns the SM-LEXICAL.

Both SM-GENERALIZATION and SM-CHILDOFGENERALIZATION are used to represent the concept of generalization of SM-ABSTRACTs.

A more careful analysis of the Supermodel shows how some of the constructs of OWL model do not have a direct correlation with its constructs. Moreover, some of the constructs for which that correspondence is present, do not contain sufficient attributes to the representation of whole information. In the following, we describe new constructs and how to extend the available ones.

Management of Intersections

To manage the intersections we introduce an sm-Set metaconstruct representing a generic set of abstracts. In order to consider also other sets (like OWL DL unions or RDF collections) we add the `Type` attribute to determine the kind of set we are considering. Elements of sm-Set (for example the classes which participate to an OWL intersection) are represented by means of an sm-ComponentOfSet metaconstruct. A set of sm-Abstract is an abstract itself so we also introduce a reference to sm-Abstract (e.g. an intersection of classes is a class itself in OWL).

Management of Restrictions

It was previously described that in the OWL model, the restrictions on object properties can be represented through the RelationshipBetweenClasses construct, while restrictions on datatype properties are dealt with the construct DatatypeProperty. The two correspondent metaconstructs in the Supermodel are respectively sm-BinaryAggregationOfAbstracts and sm-Lexical (see Figure 6.8). However, the old versions of those constructs must be revised in order to properly handle the additional information defined by the restrictions.

Recalling that a restriction in OWL is considered as a class, it is necessary to add, to the sm-BinaryAggregationOfAbstracts metaconstruct, a reference to the class that represents the restriction. Moreover, to manage the `owl:someValuesFrom` constraint we add an `AbstractAsSomeValuesFromOID` that belongs to sm-Abstract. For the sm-Lexical metaconstruct we have a similar situation except for the fact that the `owl:someValuesFrom` constraint cab be represented as a simple attribute (`someValuesFromType`).

Finally we must add a completely new metaconstruct to indicate when an sm-BinaryAggregationOfAbstracts is a restriction of a different sm-BinaryAggregationOfAbstracts (the same for two sm-Lexicals) and the kind of the restriction.

Classes Equivalence

The equivalence relation between classes can be seen as a binary relationship. For this reason, the sm-BinaryAggregationOfAbstracts metaconstruct is used to represent it within the supermodel. In this construct is added the `isEquivalence` boolean attribute that determines whether the relationship is an equivalence or not.

Properties Equivalence

The equivalence between properties (both object and datatype) is represented by the construct sm-AssertionOnProperty. In this construct we added the `isEquivalent` attribute that determines whether a property is considered equivalent to another or not.

Object and Datatype Properties Generalization

The old release of the Supermodel was able to manage only generalization between sm-Abstracts. In order to also manage the OWL properties generalizations we have enhanced the two generalization metaconstructs, i.e. sm-Generalization and sm-ChildOfGeneralization. In particular, we introduce two references in sm-Generalization, namely `ParentBinAggrOID` and `ParentLexicalOID`. The last one belongs to sm-Lexical, the other belongs to sm-BinaryAggregationOfAbstracts. Similarly we add two other references to sm-ChildOfGeneralization: that is `ChildBinAggrOID` in case of object property generalizations and `LexicalOID` in case of datatype property generalizations.

Functional Datatype Properties

In OWL we can define *functional* datatype properties. This is managed adding the boolean attribute `isFunctional` in the `sm-Lexical` metaconstruct.

Symmetric, Transitive and inverse Object Properties

Symmetric and transitive properties are easily managed in the Supermodel adding the boolean attributes `isSymmetric` and `isTransitive` to the sm-BinaryAggregationOfAbstracts metaconstruct.

For the inverse of a property we exploit the sm-AssertionOnProperty metaconstruct again, adding to it the boolean attribute `isInverse`.

SM-BinaryAggregationOfAbstract
OID
Name
isSymmetric
isTransitive
isDirected
role1
isOptional1
isFunctional1
isIdentified
role2
isOptional2
isFunctional2
isEquivalence
AbstractAsBinAggOfAbsOID
Abstract1OID
Abstract2OID
AbstractAsSomeValuesFromOID

SM-Generalization
OID
Name
isTotal
ParentAbstractOID
ParentBinAggrOID
ParentLexicalOID

SM-ChildOfGeneralization
OID
Name
GeneralizationOID
ChildAbstractOID
ChildBinAggrOID
ChildLexicalOID

SM-Set
OID
Name
type
isAnonymous
AbstractOID

SM-Abstract
OID
Name

SM-ComponentOfSet
OID
Name
SetOID
AbstractOID

SM-AssertionOnProperty
OID
Name
restrictionType
isInverse
isRestriction
isEquivalent
BinaryAggregationOfAbstractOID1
BinaryAggregationOfAbstractOID2
Lexical1OID
Lexical2OID

SM-Lexical
OID
Name
isNullable
isOptional
isIdentifier
isFunctional
type
someValuesFromType
AbstractOID
BinaryAggregationOfAbstractOID
AbstractAsLexicalOID

Figure 6.9: A portion of the extended Supermodel.

In Figure 6.9 we show the portion of interest (for ontology and database transformation) of the extended Supermodel.

6.3 From OWL Ontologies to Relational Databases

Our purpose is to translate an OWL ontology into a relational representation trying to avoid information or semantics loss. The approach translates both schemas and data: given a source instance I_OWL of a schema S_OWL expressed in the OWL data model M_OWL, and the (relational) database model M_RDB, it generates a schema S_RDB expressed in M_RDB that is "equivalent" to S_OWL and an instance I_RDB of S_RDB "equivalent" to I_OWL. A

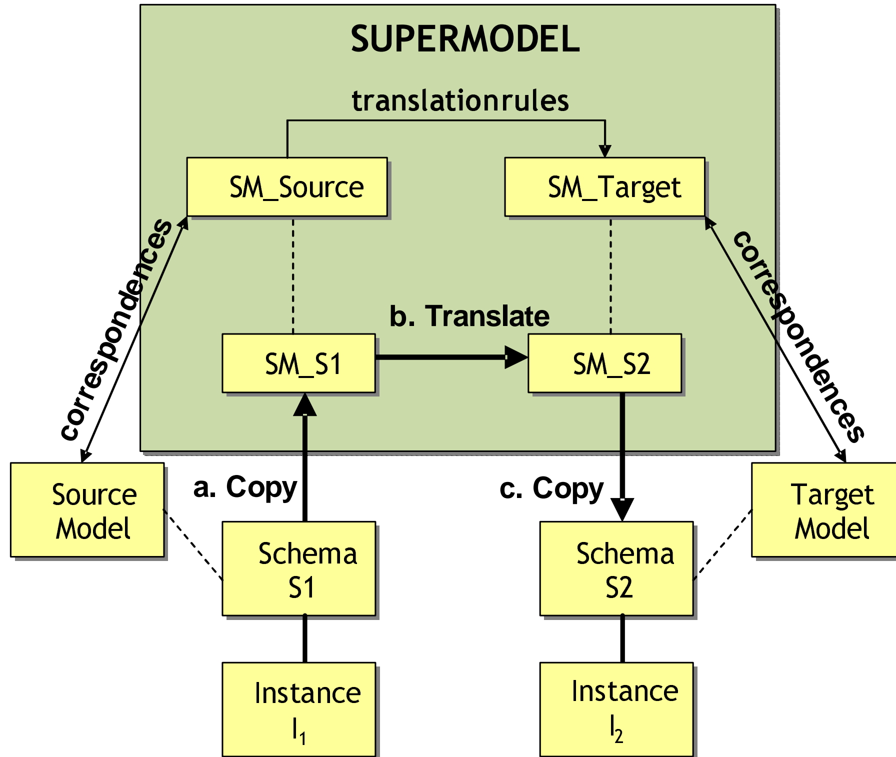


Figure 6.10: The translation process.

translation is composed of

(a) a “copy” (with construct renaming) from the source model (OWL) into the supermodel;

(b) an actual transformation within the Supermodel, whose output includes only constructs allowed in the target model (RDB);

(c) another copy (again with construct renaming) into the target model (RDB), as depicted in Figure 6.10 (in the figure we also include schema and data).

We will illustrate by a use case how the translation is performed.

Let us consider the OWL Lite example below:

```
<owl:Class rdf:ID="Employee"/>

<owl:Class rdf:ID="Project">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#managedBy"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
        1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="ProjectManager">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:ID="Employee"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#manages"/>
          <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
            1
          </owl:minCardinality>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="SoftwareDeveloper">
  <rdfs:subClassOf rdf:resource="#Employee"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#involvedIn"/>
      <owl:allValuesFrom>
        <owl:Class rdf:ID="SoftwareProject"/>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="SoftwareProject">
  <rdfs:subClassOf rdf:resource="#Project"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="involves">
  <rdfs:domain rdf:resource="#Project"/>
  <rdfs:range rdf:resource="#Employee"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="involvedIn">
  <owl:inverseOf rdf:resource="#involves"/>
  <rdfs:domain rdf:resource="#Employee"/>
  <rdfs:range rdf:resource="#Project"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="manages">
  <rdfs:domain rdf:resource="#ProjectManager"/>
  <rdfs:range rdf:resource="#Project"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="managedBy">
  <owl:inverseOf rdf:resource="#manages"/>
  <rdfs:domain rdf:resource="#Project"/>
```

```
  <rdfs:range rdf:resource="#ProjectManager"/>
</owl:ObjectProperty>

<owl:FunctionalProperty rdf:ID="SSN">
  <rdfs:domain rdf:resource="#Employee"/>
  <rdfs:range rdf:resource="&xsd;positiveInteger"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:ID="Name">
  <rdfs:domain rdf:resource="#Employee"/>
  <rdfs:range rdf:resource="&xsd;string"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<owl:DatatypeProperty rdf:ID="hobby">
  <rdfs:domain rdf:resource="#Employee"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:FunctionalProperty rdf:ID="ProjectName">
  <rdfs:domain rdf:resource="#Project"/>
  <rdfs:range rdf:resource="&xsd;string"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<owl:DatatypeProperty rdf:ID="keyword">
  <rdfs:domain rdf:resource="#Project"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

The serialization syntax chosen is RDF/XML. This simple example represents the relationships between employees and projects.

The `Employee` class is defined with the property `involvedIn`, to express that an employee can be involved in one or more projects. The `ProjectManager` class consists of all those employees who manage one `Project` at least.

For the sake of simplicity, we concentrate on schema translation omitting the details of instance translations.

Copy in the Supermodel

In order to make things concrete, we show in Figure 6.11 relational implementation of a portion of the dictionary, as we defined it in our tool, for the OWL example (the dictionary has been described in Chapter 3.2).

In this first phase, the schema, initially represented in term of OWL data model constructs, is copied in the Supermodel in terms of metaconstructs. Referring to the correspondences between the constructs of the OWL model and the Supermodel metaconstructs the result of this phase is a copy of the scheme represented by the relational dictionary shown in Figure 6.12 (we report only a simplified portion of the dictionary due to space limitation).

OWL_Class

OID	Name
c1	Employee
c2	Project
c3	ProjectManager
c4	SoftwareDeveloper
c5	SoftwareProject
c6	RestrictionClass1
c7	RestrictionClass2
c8	RestrictionClass3
c9	IntersectionClass1

OWL_Intersection

OID	Name	Class
in1	Intersection	c9

OWL_ComponentOfIntersection

OID	Name	Intersection	Class
ci1	CompRestr1	in1	c1
ci2	CompRestr2	in1	c7

OWL_DatatypeProperty

OID	Name	Type	isOpt	isFunc	Class	ClassAsRestr
dp1	SSN	int	true	true	c1	-
dp2	Name	string	true	true	c1	-
dp3	Hobby	string	true	false	c1	-
dp4	Keyword	string	true	false	c2	-

OWL_RelationshipBetweenClasses

OID	Name	isSymm	isTrans	isDirect	isOpt1	isFunc1	isOpt2	isFunc2	isEquiv	Class1	Class2	ClassAsRestr
r1	involves	false	false	true	true	false	true	false	false	c2	c1	-
r2	involvedIn	false	false	true	true	false	true	false	false	c1	c2	-
r3	manages	false	false	true	true	false	true	false	false	c1	c2	-
r4	managedBy	false	false	true	true	true	true	false	false	c2	c1	-
r5	managedByRestr	false	false	true	false	true	true	false	false	c2	c1	c6
r6	managesRestr	false	false	true	false	false	true	false	false	c1	c2	c7
r7	involvedInRestr	false	false	true	false	false	true	false	false	c1	c5	c8
r8	equivalence	false	false	false	false	false	false	false	true	c3	c9	-

OWL_AssertionOnProperty

OID	Name	isRestriction	isInverse	isEquivalent	restrictionType	Rel1	Rel2	Datatype1	Datatype2
as1	Assertion1	true	false	false	cardinality	r5	r4	-	-
as2	Assertion2	true	false	false	minCardinality	r6	r3	-	-
as3	Assertion3	true	false	false	allValuesFrom	r7	r2	-	-
as4	Assertion4	false	true	false	-	r2	r1	-	-
as5	Assertion5	false	true	false	-	r4	r3	-	-

OWL_Generalization

OID	Name	ParentClass	ParentObjProp	ParentDataProp
g1	Gen1	c6	-	-
g2	Gen2	c1	-	-
g3	Gen3	c8	-	-
g4	Gen4	c2	-	-

OWL_ChildOfGeneralization

OID	Name	Gen	ChildClass	ChildObjPr	ChildDataPr
ch1	Child1	g1	c2	-	-
ch2	Child2	g2	c4	-	-
ch3	Child3	g3	c4	-	-
ch4	Child4	g4	c5	-	-

Figure 6.11: Dictionary Tables for the OWL Example.

SM_Abstract

OID	Name
a1	Employee
a2	Project
a3	ProjectManager
a4	SoftwareDeveloper
a5	SoftwareProject
a6	RestrictionClass1
a7	RestrictionClass2
a8	RestrictionClass3
a9	IntersectionClass1

SM_Set

OID	Name	Abs
s1	Intersection	a9

SM_ComponentOfSet

OID	Name	Set	Class
cs1	CompRestr1	s1	a1
cs2	CompRestr2	s1	a7

SM_Lexical

OID	Name	Type	isOpt	isFunc	Abs	AbsAsLexical
L1	SSN	int	true	true	a1	-
L2	Name	string	true	true	a1	-
L3	Hobby	string	true	false	a1	-
L4	Keyword	string	true	false	a2	-

SM_BinaryAggregationOfAbstracts

OID	Name	isSymm	isTrans	isDirect	isOpt1	isFunc1	isOpt2	isFunc2	isEquiv	Abs1	Abs2	AbsAsBinAggr
ba1	involves	false	false	true	true	false	true	false	false	a2	a1	-
ba2	involvedIn	false	false	true	true	false	true	false	false	a1	a2	-
ba3	manages	false	false	true	true	false	true	false	false	a1	a2	-
ba4	managedBy	false	false	true	true	true	true	false	false	a2	a1	-
ba5	managedByRestr	false	false	true	false	true	true	false	false	a2	a1	a6
ba6	managesRestr	false	false	true	false	false	true	false	false	a1	a2	a7
ba7	involvedInRestr	false	false	true	false	false	true	false	false	a1	a5	a8
ba8	equivalence	false	false	false	false	false	false	false	true	a3	a9	-

SM_AssertionOnProperty

OID	Name	isRestriction	isInverse	isEquivalent	restrictionType	BinAggr1	BinAggr2	Lex1	Lex2
as1	Assertion1	true	false	false	cardinality	ba5	ba4	-	-
as2	Assertion2	true	false	false	minCardinality	ba6	ba3	-	-
as3	Assertion3	true	false	false	allValuesFrom	ba7	ba2	-	-
as4	Assertion4	false	true	false	-	ba2	ba1	-	-
as5	Assertion5	false	true	false	-	ba4	ba3	-	-

SM_Generalization

OID	Name	ParentAbs	ParentBinAggr	ParentLexical
g1	Gen1	a6	-	-
g2	Gen2	a1	-	-
g3	Gen3	a8	-	-
g4	Gen4	a2	-	-

SM_ChildOfGeneralization

OID	Name	Gen	ChildAbs	ChildBinAggr	ChildLex
ch1	Child1	g1	a2	-	-
ch2	Child2	g2	a4	-	-
ch3	Child3	g3	a4	-	-
ch4	Child4	g4	a5	-	-

Figure 6.12: A portion of the Supermodel dictionary.

After the first "copy" operation, we have obtained database tables that fully describe the ontology structure, exploiting a logical organization that reflects the constructs of the ontology language. Once the ontology is translated in terms of a relational representation, it can be queried, modified and converted back to the source ontology language. Since all the characteristics of the constructs used to define the source ontology are stored into the relational representation, it is possible to perform the reverse transformation from the relational representation back to the original ontology. Moreover, our meta-representation of an OWL ontology can subsequently be used to perform translations to other formalisms.

Translation within the Supermodel

The "real" transformation is made within the Supermodel as previously stated. The main objective is to perform the translation reducing the information loss at minimum.

The flexibility of our approach allows the user to choose the form of the translation, defining how to generate the target model. For example, it is possible to define a translation where a relation is generated for each kind of class and another translation where only the named classes are transformed in relations. Furthermore, generalization can be mapped to the relational model in many ways and all the choices are available to the user.

In the following, we describe the different steps of a chosen translation that are realized through the application of convenient Datalog rules.

Step 1. In this step, we identify the *Named* Abstracts, that represent those SM-ABSTRACTs belonging to OWL named classes. More precisely, let A an SM-ABSTRACT. It is also a *Named* Abstract if:

- A OID does not appear as value of `AbstractAsBinAggrOfAbsOID` attribute of any SM-BINARYAGGREGATIONOFABSTRACTS metaconstructs;
- A OID does not appear as value of `AbstractAsLexicalOID` attribute of any SM-LEXICAL;
- A OID does not appear as value of `AbstractOID` attribute of any SM-SET;

In the example `Employee`, `Project`, `ProjectManager`, `SoftwareDveloper` and `SoftwareProject` classes belong to *Named* Abstract family.

BinAggrOfAbs	isFunctional1	isFunctional2	Abstract1OID	Abstract2OID
BA2	true	false	A1	A2
BA1	false	true	-	A3

⇓

BinAggrOfAbs	isFunctional1	isFunctional2	Abstract1OID	Abstract2OID
BA2	true	false	A1	A2
BA1	true	true	A1	A3

Figure 6.13: The inheritance process.

Step 2. Let be BAA an SM-BINARYAGGREGATIONOFABSTRACTS and BAA_G a generalization of BAA, therefore BAA inherits some attributes values from BAA_G; namely they are (we consider only the attributes that are relevant in this context):

- `isFunctional1` that indicates if the relation is functional or not;
- `isFunctional2` that indicates if the relation is inverse functional or not;
- Abstract1OID that indicated the domain of the relation
- Abstract2OID that indicates the range of the relation

The situation of the inheritance process is shown in Figure 6.13, where $BA2$ is the generalization of $BA1$. We remark that the same process can be recursively applied if there are more generalizations (e.g. $BA3$ is the generalization of $BA2$ that is a generalization of $BA1$).

Step 3. As for the previous step there can be also generalization of SM-LEXICAL metaconstructs. We adopt in this case a similar inheritance process.

Step 4. In this step the SM-ABSTRACTs that are subsumed in another SM-ABSTRACT are identified. Let be A_1 and A_2 two SM-ABSTRACTs. A_1 is subsumed in A_2 if:

- there exists an SM-GENERALIZATION that explicitly relates A_1 and A_2 or
- there are an SM-GENERALIZATION that relates A_1 and an SM-ABSTRACT that is an intersection in which one element is A_2 or

- there is an equivalence relation between A_1 and an SM-ABSTRACT that is an intersection in which one element is A_2. We remark that the equivalence relation is specified by means of a convenient SM-BINARYAGGREGATIONOFABSTRACTS with the `isEquivalence` attribute set to `TRUE`.

With regard to the considered example, it appears that SM-ABSTRACTS `ProjectManager` and `SoftwareDeveloper` are subsumed in `Employee` and `SoftwareProject` is subsumed in `Project`.

Step 5. This step is devoted to the identification of the restricted binary relations. A restricted binary relation RBA_1 is a binary relation between SM-ABSTRACTS with constraints in `isOptional` and `isFunctional` attributes and `Abstract2OID` reference.

A restricted SM-BINARYAGGREGATIONOFABSTRACTS is associated with an SM-ABSTRACT, composed of all those instances involved in the restricted relation. In this context we define the concept of *owner* of a restricted relation (that we simply call restriction).

A named SM-ABSTRACT A is an owner of a restriction that involves the SM-ABSTRACT RA in the following situations:

- there exists an equivalence relation between A and RA.
- there exists an SM-GENERALIZATION from A to RA
- there exists an equivalence relation between A and an SM-ABSTRACT that is an intersection in which one element is RA.
- there exists an SM-GENERALIZATION that relates A and an SM-ABSTRACT that is an intersection in which one element is RA.

Referring to the example we can say that SOFTWAREDEVELOPER is the owner of the restriction INVOLVEDINRESTRICTED, which in turn is a restriction of the INVOLVEDIN relation. PROJECTMANAGER also *owns* the restriction MANAGESRESTRICTED defined on the MANAGES relation.

Then we can identify when a named SM-ABSTRACT A assumes the role of domain of a relation RA. The possibilities are:

- A is explicitly defined as a domain of BA
- A owns a binary relation RBA defined on BA

Domain Abstract	BinAggrOfAbstract
Employee	involvedIn
ProjectManager	manages
SoftwareDeveloper	involvedIn
Project	involves
Project	managedBy

Figure 6.14: Results of step 5.

Referring to the example we can report the results of this step in Figure 6.14.

Step 6. We define in this step the characteristics of each SM-ABSTRACT derived from the previous step.

Let be C an SM-ABSTRACT that is subsumed in the SM-ABSTRACT B that is, in turn, subsumed in the SM-ABSTRACT A. Let also be true the following conditions:

- A is related to the SM-BINARYAGGREGATIONOFABSTRACTS BA with the `isFunctional` attribute set to FALSE;
- B has a domain role in the relation BA
- C has a domain role in the relation BA

The situation is shown in Figure 6.15, which also refers to the example.

Step 7 and 8. These steps are similar to steps 5 and 6, but we refer to SM-LEXICALs instead of SM-BINARYAGGREGATIONOFABSTRACTS.

Step 9. This is the first "real" transformation step, which exploits the information extracted in the previous steps. In this step the *Named* SM-ABSTRACTS, identified in Step 1, will be transformed into different SM-AGGREGATIONS. For each SM-AGGREGATION also creates an SM-LEXICAL.

Referring to the example, EMPLOYEE is translated in the homonymous SM-AGGREGATION and also the SM-LEXICAL `EmployeeId` will be created.

Step 10. The SM-BINARYAGGREGATIONOFABSTRACTSs are translated. Considering a binary relation with domain D_1 and range R_1 it is translated

Abstract	BinAggrOfAbs	isOptional1	isFunctional1	Abs2OID
A	BA	true	false	D
B	BA	false	false	D
C	BA	false	true	D

Domain Abstract	BinAggrOfAbs	isOptional1	isFunctional1	Abs2OID
Employee	involvedIn	true	false	Project
SoftwareDeveloper	involvedIn	true	false	SoftwareProject
ProjectManager	manages	false	false	Project
Project	involves	true	false	Employee
Project	managedBy	false	true	ProjectManager

Figure 6.15: Results of step 6.

in an SM-LEXICAL belonging to an SM-AGGREGATION that corresponds to the SM-ABSTRACT D_1. This SM-LEXICAL is also involved in an SM-FOREIGNKEY that belongs to an SM-AGGREGATION corresponding to R_1.

It should be noted that in case of two binary relations, which is one inverse of the other, only one is transformed, while the other is not considered. This is to avoid the creation of two cross-references between SM-AGGREGATIONS.

Step 11. In this step we exploit the results of Step 4. If an SM-ABSTRACT A is subsumed in an SM-ABSTRACT B, an SM-FOREIGNKEY belonging to B is created.

The application of the aforementioned translation steps causes the generation, within the Supermodel, of a new schema defined in terms of metaconstructs that is compatible with the relational data model, as depicted in Figure 6.16.

From Supermodel To Relational Data Model

In this third and final phase we perform a copy of the schema generated from the previous translation, in a schema defined in terms of the relational model constructs.

SM_Aggregation

OID	Name
ag1	Employee
ag2	Project
ag3	ProjectManager
ag4	SoftwareDeveloper
ag5	SoftwareProject
ag6	EmpInvolvedInProject
ag7	SoftwDevInvolvedInSoftProj
ag8	HobbyValue
ag9	Keywordvalue

SM_ForeignKey

OID	Name	AggrFrom	x
fk1	ForeignKey1	ag2	ag3
fk2	ForeignKey2	ag6	ag1
fk3	ForeignKey3	ag6	ag2
fk4	ForeignKey4	ag7	ag4
fk5	ForeignKey5	ag7	ag5
fk6	ForeignKey6	ag8	ag1
fk7	ForeignKey7	ag9	ag2
fk8	ForeignKey8	ag3	ag1
fk9	ForeignKey9	ag4	ag1
fk10	ForeignKey10	ag5	ag2

SM_ComponentOfForeignKey

OID	ForeignKey	Name	LexicalFrom	LexicalTo
cfk1	fk1	Component1	l6	l7
cfk2	fk2	Component2	l10	l1
cfk3	fk3	Component3	l11	l4
cfk4	fk4	Component4	l12	l8
cfk5	fk5	Component5	l13	l9
cfk6	fk6	Component6	l14	l1
cfk7	fk7	Component7	l16	l4
cfk8	fk8	Component8	l7	l1
cfk9	fk9	Component9	l8	l1
cfk10	fk10	Component10	l9	l4

SM_Lexical

OID	Aggregation	Name	Type	isIdentifier	isNullable
l1	ag1	EmployeeId	int	true	false
l2	ag1	SSN	int	false	true
l3	ag1	Name	string	false	true
l4	ag2	ProjectId	int	true	false
l5	ag2	ProjectName	string	false	true
l6	ag2	ManagedBy	int	false	false
l7	ag3	ProjectManagerId	int	true	false
l8	ag4	SoftwareDeveloperId	int	true	false
l9	ag5	SoftwareProjectId	int	true	false
l10	ag6	Employee	int	true	false
l11	ag6	Project	int	true	false
l12	ag7	SoftwareDeveloper	int	true	false
l13	ag7	SoftwareProject	int	true	false
l14	ag8	Employee	int	true	false
l15	ag8	Value	string	true	false
l16	ag9	Project	int	true	false
l17	ag9	Value	string	true	false

Figure 6.16: Translation within the Supermodel.

The results of the transformation for the considered example are shown in the relational dictionary of Figure 6.17, therefore the resulting relational schema is depicted in Figure 6.18.

6.4 From Relational Databases to OWL ontologies

The purpose is to translate a relational database schema into an OWL-compliant one. The three main phases are the same ones described in Section 6.3 and also in this case we refer to an example. The relational schema to be transformed is shown in Figure 6.19.

Copy in the Supermodel

As for the inverse translation, the first operation is substantially a copy of the schema in the Supermodel by means of metaconstructs. The results for the proposed example are shown in Figure 6.20.

Translation within the Supermodel

Also in this case the translation is made by means of metaconstructs transformation within the Supermodel.

In particular we describe the different steps of the translation that are realized through the application of our Datalog-like rules.

Step 1. This step is devoted to the translation of the SM-AGGREGATION metaconstruct that does not have direct correspondences with OWL data model construct.

An SM-AGGREGATION is generally translated into an SM-ABSTRACT with the same name. In our example, this is the situation of STUDENT, WORKERSTUDENT, COURSE, EXAM and PROFESSOR.

If the SM-AGGREGATION is related with two SM-LEXICAL that are also related to an SM-FOREIGNKEY, we have to perform the following operations:

- an SM-BINARYAGGREGATIONOFABSTRACTS BA_1 is created to represent the relation between the SM-ABSTRACTS;
- an SM-BINARYAGGREGATIONOFABSTRACTS BA_2;

Rel_Table

OID	Name
t1	Employee
t2	Project
t3	ProjectManager
t4	SoftwareDeveloper
t5	SoftwareProject
t6	EmpInvolvedInProject
t7	SoftwDevInvolvedInSoftProj
t8	HobbyValue
t9	Keywordvalue

Rel_ForeignKey

OID	Name	AggrFrom	AggrTo
fk1	ForeignKey1	t2	t3
fk2	ForeignKey2	t6	t1
fk3	ForeignKey3	t6	t2
fk4	ForeignKey4	t7	t4
fk5	ForeignKey5	t7	t5
fk6	ForeignKey6	t8	t1
fk7	ForeignKey7	t9	t2
fk8	ForeignKey8	t3	t1
fk9	ForeignKey9	t4	t1
fk10	ForeignKey10	t5	t2

Rel_ComponentOfForeignKey

OID	ForeignKey	Name	LexicalFrom	LexicalTo
cfk1	fk1	Component1	c6	c7
cfk2	fk2	Component2	c10	c1
cfk3	fk3	Component3	c11	c4
cfk4	fk4	Component4	c12	c8
cfk5	fk5	Component5	c13	c9
cfk6	fk6	Component6	c14	c1
cfk7	fk7	Component7	c16	c4
cfk8	fk8	Component8	c7	c1
cfk9	fk9	Component9	c8	c1
cfk10	fk10	Component10	c9	c4

Rel_Column

OID	Aggregation	Name	Type	isIdentifier	isNullable
c1	t1	EmployeeId	int	true	false
c2	t1	SSN	int	false	true
c3	t1	Name	string	false	true
c4	t2	ProjectId	int	true	false
c5	t2	ProjectName	string	false	true
c6	t2	ManagedBy	int	false	false
c7	t3	ProjectManagerId	int	true	false
c8	t4	SoftwareDeveloperId	int	true	false
c9	t5	SoftwareProjectId	int	true	false
c10	t6	Employee	int	true	false
c11	t6	Project	int	true	false
c12	t7	SoftwareDeveloper	int	true	false
c13	t7	SoftwareProject	int	true	false
c14	t8	Employee	int	true	false
c15	t8	Value	string	true	false
c16	t9	Project	int	true	false
c17	t9	Value	string	true	false

Figure 6.17: The resulting relational dictionary.

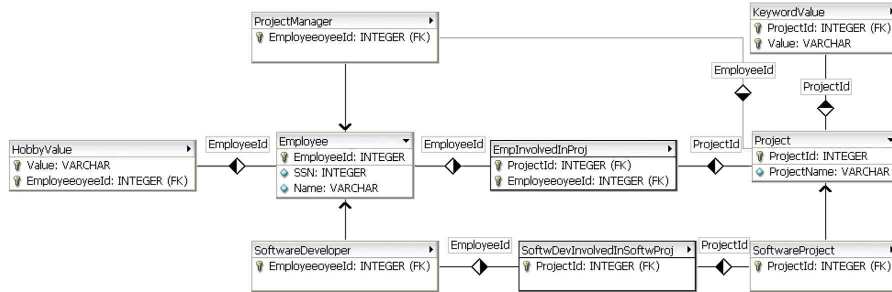


Figure 6.18: The resulting relational schema.

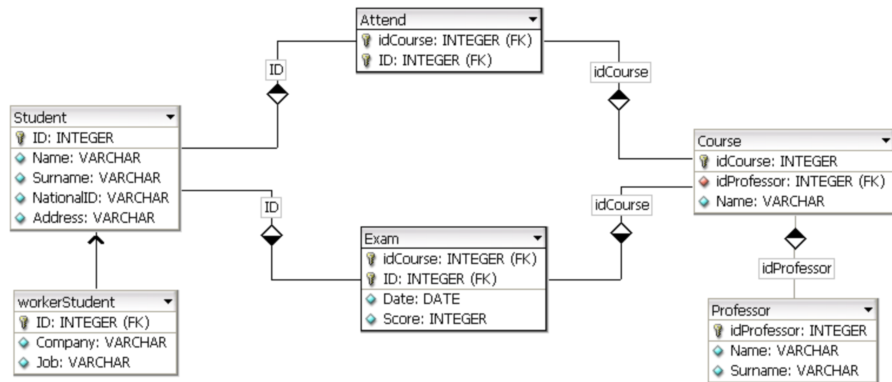


Figure 6.19: The relational schema to be translated.

- it is necessary that BA_2 is inverse of BA_1. To represent this property an SM-ASSERTIONONPROPERTY is created.

This is the situation of ATTEND.

Step 2. If an SM-AGGREGATION A_1 (translated into SM-ABSTRACT A_1) is related to an SM-LEXICAL that is an identifier of A_1 and is also a component of a foreign key to another SM-AGGREGATION A_2 (translated into SM-ABSTRACT A_2), then a SM-GENERALIZATION between A_2 and A_1 (SM-CHILDOFGENERALIZATION) is created.

SM_Aggregation	
OID	Name
ag1	Student
ag2	Professor
ag3	Course
ag4	Attend
ag5	Exam
ag6	WorkerStudent

SM_ForeignKey			
OID	Name	AggrFrom	AggrTo
fk1	Foreign Key 1	ag4	ag1
fk2	Foreign Key 2	ag4	ag3
fk3	Foreign Key 3	ag3	ag2
fk4	Foreign Key 4	ag5	ag1
fk5	Foreign Key 5	ag5	ag3
fk6	Foreign Key 6	ag6	ag1

SM_Lexical					
OID	Aggregation	Name	Type	isIdentifier	isNullable
l1	ag1	ID	int	true	false
l2	ag1	Name	string	false	false
l3	ag1	Surname	string	false	false
l4	ag1	NationalID	string	false	false
l5	ag1	Address	string	false	true
l6	ag2	idProfessor	int	true	false
l7	ag2	Name	string	false	false
l8	ag2	Surname	string	false	false
l10	ag3	idCourse	int	true	false
l11	ag3	Name	string	false	false
l12	ag3	idProfessor	int	false	false
l13	ag4	ID	int	true	false
l14	ag4	idCourse	int	true	false
l15	ag5	ID	int	true	false
l16	ag5	idCourse	int	true	false
l17	ag5	Date	string	false	false
l18	ag5	Score	int	false	false
l19	ag6	ID	int	true	false
l20	ag6	Company	string	false	false
l21	ag6	Job	string	false	true

SM_ComponentOfForeignKey				
OID	ForeignKey	Name	LexicalFrom	LexicalTo
cfk1	fk1	Component 1	l13	l1
cfk2	fk2	Component 2	l14	l10
cfk3	fk3	Component 3	l12	l6
cfk4	fk4	Component 4	l15	l1
cfk5	fk5	Component 5	l16	l10
cfk6	fk6	Component 6	l19	l1

Figure 6.20: The "copy" of the relational schema in the Supermodel.

This is the case of STUDENT and WORKERSTUDENT.

Step 3. For each SM-LEXICAL L that belongs to an SM-FOREIGNKEY we must perform the following operations:

- an SM-BINARYAGGREGATIONOFABSTRACTS BA_1 is created between the SM-ABSTRACT that belongs to the SM-AGGREGATION that is related to L and the SM-ABSTRACT that represents the foreign key;
- an SM-BINARYAGGREGATIONOFABSTRACTS BA_2;
- an SM-ASSERTIONONPROPERTY that defines the inverse relation between B_2 and B_1 is created.

In the example, the elements involved in this step are PROFESSORID of COURSE and ID and COURSEID of EXAM.

Step 4. Each SM-LEXICAL we consider, that does not belong to a particular SM-COMPONENTOFFOREIGNKEY is translated into another SM-LEXICAL with the attributes `isOptional` and `isFunctional` set to `TRUE`.

Step 5. In this step the cardinality constraints are added to both the SM-BINARYAGGREGATIONOFABSTRACTSs and SM-LEXICALs metaconstructs that have been created in the previous steps. More precisely, for each SM-LEXICAL of the source schema that are defined as *not-nullable* we have the two following possibilities:

- if an SM-LEXICAL L has been translated, as described in Step 3, into an SM-BINARYAGGREGATIONOFABSTRACTS BA_1:
  - an SM-BINARYAGGREGATIONOFABSTRACTS BA_2 that represents the inverse of BA_1 is created;
  - to represent the inverse property an SM-ASSERTIONONPROPERTY is created;
  - an SM-ABSTRACT R that represents a restriction that contains all instances that belongs to BA_2 is created;
  - an SM-GENERALIZATION between R and the SM-ABSTRACT derived from the SM-AGGREGATION related to L is created.
- if an SM-LEXICAL L has been translated, as described in Step 4, into another SM-LEXICAL L_1:
  - an SM-LEXICAL L_2 that is a restriction of L_1, with the same characteristics is created;

SM_Abstract

OID	Name
a1	Student
a2	Professor
a3	Course
a4	Exam
a5	WorkerStudent
a6	RestrictionClass1
a7	RestrictionClass2
a7	RestrictionClass3
...	...

SM_Lexical

OID	Name	Type	isOpt	isFunc	Abs	AbsAsLex
l1	ID	int	true	true	a1	-
l2	Name	string	true	true	a1	-
l3	Surname	string	true	true	a1	-
l4	Address	string	true	true	a1	-
...	...	...	...	...	...	...
l5	IDRestricted	int	false	true	a1	a6
l6	NameRestricted	string	false	true	a1	a7

SM_BinaryAggregationOfAbstract

OID	Name	isSymm	isTrans	isDirect	isOpt1	isFunc1	isOpt2	isFunc2	isEquiv	Abs1	Abs2	AbsAsBinAggr
ba1	Attend	false	false	true	true	false	true	false	false	a1	a3	-
ba2	InvAttend	false	false	true	true	false	true	false	false	a3	a1	-
ba3	idProfessor	false	false	true	true	true	true	false	false	a3	a2	-
...	...	...	...	...	...	...	...	...	...	...	...	...
ba4	idProfessorRestr	false	false	true	false	true	true	false	false	a3	a2	a8

SM_AssertionOnProperty

OID	Name	isRestriction	isInverse	isEquivalent	restrictionType	BinAggr1	BinAggr2	Lexical1	Lexical2
as1	Assertion1	false	true	false	-	ba2	ba1	-	-
as2	Assertion2	true	false	false	minCardinality	ba4	ba3	-	-
as3	Assertion3	true	false	false	minCardinality	-	-	l5	l1
as4	Assertion4	false	true	false	minCardinality	-	-	l6	l2
...	...	...	...	...	...	...	...	...	...

SM_Generalization

OID	Name	ParentAbs	ParentBinAggr	ParentLexical
g1	Gen1	a1	-	-
g2	Gen2	a6	-	-
g3	Gen3	a7	-	-
g4	Gen4	a8	-	-
...	...	...	...	...

SM_ChildOfGeneralization

OID	Name	Generaliz	ChildAbs	ChildBinAggr	ChildLexical
ch1	Child1	g1	a5	-	-
ch2	Child2	g2	a1	-	-
ch3	Child3	g3	a1	-	-
ch4	Child4	g4	a3	-	-
...	...	...	...	...	...

Figure 6.21: The result of relational schema translation in the Supermodel.

- an SM-ASSERTIONONPROPERTY is created to state that L_2 is a restriction of minimum cardinality on L_1;
- an SM-ABSTRACT R that represents a restriction that contains all instances that belongs to L_2 is created;
- an SM-GENERALIZATION between R and the SM-ABSTRACT derived from the SM-AGGREGATION related to L is created.

At the end of the aforementioned step we have, in the Supermodel, a situation similar to the one shown in Figure 6.21 (we omit the details due to the lack of space).

OWL_Class	
OID	Name
c1	Student
c2	Professor
c3	Course
c4	Exam
c5	WorkerStudent
c6	RestrictionClass1
c7	RestrictionClass2
c8	RestrictionClass3
...	...

OWL_DatatypeProperty						
OID	Name	Type	isOpt	isFunc	Class	ClassAsRestr
dp1	ID	int	true	true	c1	-
dp2	Name	string	true	true	c1	-
dp3	Surname	string	true	true	c1	-
dp4	Address	string	true	true	c1	-
...	...	...	...	...	...	...
dp5	IDRestricted	int	false	true	c1	c6
dp6	NameRestricted	string	false	true	c1	c7

OWL_RelationshipBetweenClasses												
OID	Name	isSymm	isTrans	isDirect	isOpt1	isFunc1	isOpt2	isFunc2	isEquiv	Class1	Class2	ClassAsRestr
r1	Attend	false	false	true	true	false	true	false	false	c1	c3	-
r2	InvAttend	false	false	true	true	false	true	false	false	c3	c1	-
r3	idProfessor	false	false	true	true	true	true	false	false	c3	c2	-
...	...	...	...	...	...	...	...	...	...	...	...	...
r4	idProfRestr	false	false	true	false	true	true	false	false	c3	c2	c8

OWL_AssertionOnProperty									
OID	Name	isRestriction	isInverse	isEquivalent	restrictionType	Rel1	Rel2	Datatype1	Datatype2
as1	Assertion1	false	true	false	-	r2	r1	-	-
as2	Assertion2	true	false	false	minCardinality	r4	r3	-	-
as3	Assertion3	true	false	false	minCardinality	-	-	dp5	dp1
as4	Assertion4	false	true	false	minCardinality	-	-	dp6	dp2
...	...	...	...	...	...	...	...	...	...

OWL_Generalization				
OID	Name	ParentClass	ParentObjProp	ParentDataProp
g1	Gen1	c1	-	-
g2	Gen2	c6	-	-
g3	Gen3	c7	-	-
g4	Gen4	c8	-	-
...	...	...	...	...

OWL_ChildOfGeneralization					
OID	Name	Generaliz	ChildClass	ChildObjProp	ChildDataProp
ch1	Child1	g1	c5	-	-
ch2	Child2	g2	c1	-	-
ch3	Child3	g3	c1	-	-
ch4	Child4	g4	c3	-	-
...	...	...	...	...	...

Figure 6.22: Dictionary tables of resulting OWL schema.

From Supermodel to OWL Data Model

The last phase of the translation process is a “copy” operation that produces the target schema using the constructs of the OWL model. The process is the same of the inverse translation (from relational to OWL) and the results are shown in Figure 6.22 (only a portion of the dictionary is presented).

6.5 Information Loss

From Database To Ontology

As many constraints, relationships and other semantics in relational data-base are implicit, or even lacking, the ontologies mapped from relational model are maybe not complete in semantics.

However, thanks to the characteristics of our approach, we have minimum loss of information at the end of the translation. A typical example is the database primary key concept that can not be properly represented in ontologies. We avoid this problem exploiting the inverse functional properties of OWL. Properties may be stated to be inverse functional. If a property is inverse functional then the inverse of the property is functional. Thus the inverse of the property has at most one value for each individual.

We can exploit this characteristic to refer to an unambiguous property. For example, `hasITTaxCode` (a unique tax identifier for Italy residents) may be stated to be inverse functional (or unambiguous). The inverse of this property (which may be referred to as `isTheTaxCodeFor`) has at most one value for any individual in the class of tax code numbers. Thus any one person's tax code number is the only value for their `isTheTaxCodeFor` property. From this a reasoner can deduce that two different individual instances of Person can not have the identical IT Tax Code Number. Also, a reasoner can deduce that if two instances of Person have the same Tax Code Number, then those two instances refer to the same individual.

In this way we can simulate the concept of primary key also in OWL ontologies.

From Ontology To Database

Translating from a more expressive model (ontology) to a less expressive one (database) always involves a loss of information. In our approach we try to reduce this loss at minimum, at the same time, some dynamical aspects in relational model, such as triggers, storage procedure cannot be mapped. In particular we are taking into account the following information.

someValuesFrom Restrictions

`someValuesFrom` is stated on a property with respect to a class. A particular class may have a restriction on a property that at least one value for that property is of a certain type. More precisely, it defines a property that should

have at least one value of a certain type, but also could have other values, possibly of different types. Obviously, this is incompatible with the relational model, in which the type of a column must be unique and defined at the time of its creation. Consequently, while the other types of restrictions are considered in the transformation process, we choose to ignore `someValuesFrom` restrictions at the moment.

Equivalence

In OWL it is possible to define two classes as *equivalent*, in other terms, which have the same instances. In the relational model it is not possible to set up two tables as equivalent, so this type of information is not taken into account during translation process, except for some particular cases. For example, the equivalence between a named class and a class defined by a restriction on a certain property is managed considering the named class as the owner of the restricted property. Moreover, in case of equivalence between a class C and a class composed of the intersection of classes C_1 and C_2 we consider that C is subclass of both C_1 and C_2 creating fictitious generalizations. Therefore we apply the aforementioned methodology of translation.

In the same way we don't model the equivalence relation between properties.

Properties Generalization

The concept of subclass is represented in a relational schema through a foreign key defined in the table corresponding to the child class and referred to the table corresponding to the parent class.

On the other side the concept of subproperty can not be represented in the relational model. This is because during the transformation we translate OWL properties in relational tables or columns, losing the generalization information.

6.6 Discussion

In this chapter we described the extension of our approach to manage data and schema translation between OWL ontologies and (relational) databases. We have described also how the MIDST supermodel can be extended to represent the OWL model and the needed rules to perform the translation in both directions.

Because many constraints, relationships and other semantics are implicit, or even lacking, the translations are not always "complete". However exploiting

the metamodel approach we can provide convenient translations with minimum information loss, as demonstrated by our experiments.

Chapter 7

A Tool Supporting Semantic Annotation Interoperability

In this chapter we present our customizable and extensible tool to implement ModelGen, the model management operator that translates data and schema from one model to another. The approach is interesting because the tool exposes the dictionary that stores models, schemas, and the rules used to implement translations. In this way, the transformations can be customized and the tool can be easily extended.

7.1 The MIDST Tool

We developed a tool to validate the concepts in previous chapters and to test their effectiveness. The main parts of the MIDST tool are the generic data dictionary, the rule repository and a Java plug-in-based application that handles the components in a modular way. The main components include a set of modules to support users in defining and managing models, schemas, Skolem functions, translations, import and export of schemas, extraction of signatures from rules (see [AGC08]) and models and generation of translation plans.

The tool offers functions for three categories of users, corresponding to three different levels of expertise. Namely they are (from less expert to more expert):

- The *designer* - can define or import/export schemas for available models and perform translations over them.

- The *model engineer* - can define new models by using the available metaconstructs.
- The *metamodel engineer* can add new metaconstructs to the metamodel and define translation rules for them; in this way she can extend the set of models handled by the system.

All of the above activities can be done without touching the tools source code. The definition of a model or of a schema involves populating tables of the dictionary, whereas the definition of translations involves inserting elements in the rule repository.

Let us describe the tool by showing how the main activities are performed. First of all we describe the typical activity of the model engineer (MoE), the definition of a model. This is done by "creating" a new model, giving it a name, and then specifying its constructs. This latter activity is the more interesting one, and it is done (interactively) in two main steps: (i) choosing a metaconstruct from a pop-up menu and giving it a name within the model, and (ii) adding the desired properties available for the chosen metaconstruct. For example, suppose the user is creating a version of the RDF model. The *MoE*, in order to define the first construct, will probably specify that he wants to add a construct corresponding to the `Abstract` metaconstruct `RDFNode` (see Section 4.3) as shown in Figure 7.1.

During the definition process, the *MoE* can add, remove, and alter constructs and construct properties. When the model is complete, the user requests a finalization, during which the system creates the corresponding dictionary structure. It is always possible to have a graphical representation of the model through our visualization tool as shown in Figure 7.2.

Then we can describe the operation of building schemas through an interactive interface, that is usually performed by designers (De). After choosing the model, *De* can define the various elements, one at the time, by choosing a construct of the model and then giving a name and the associated properties and references, if needed. For instance, the *De* can define `Student` and `Person` corresponding to the CLASS construct and then he can define that `Student` is a restriction of `Person` through the RELATIONSHIPBETWEENCLASSES construct. In Figure 7.3 the last of these steps is presented.

The interactive definition has been useful for testing elementary steps (or for changes to existing schemas), but it would not be effective in practical settings. Therefore, as a major option,we have developed an import (and export) module. It relies on a persistence manager for the supermodels constructs.

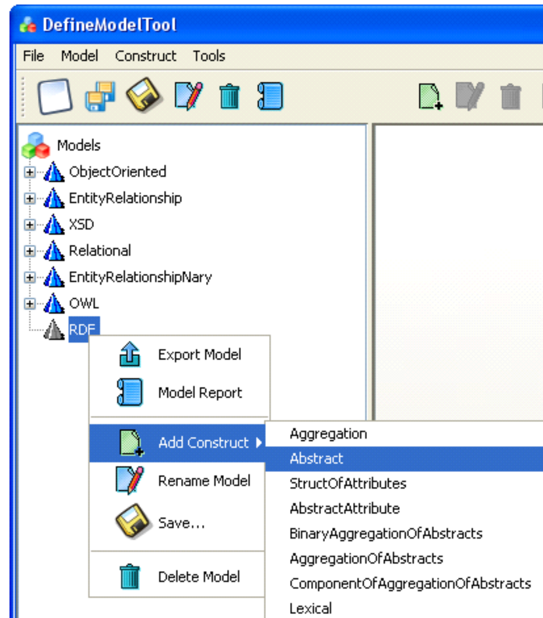


Figure 7.1: Creation of a new construct in a model.

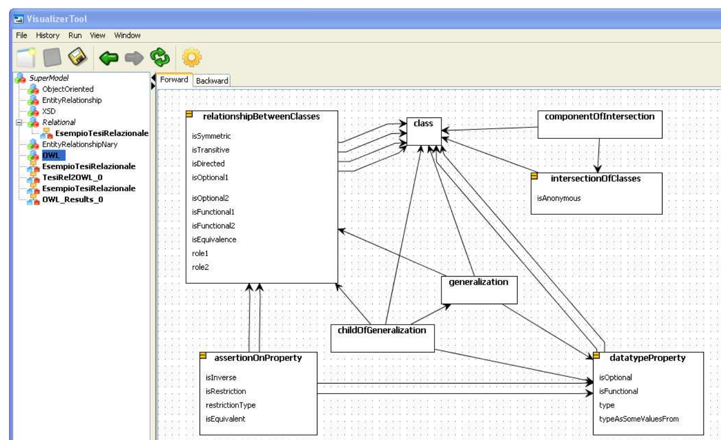


Figure 7.2: The visualization of the OWL model.

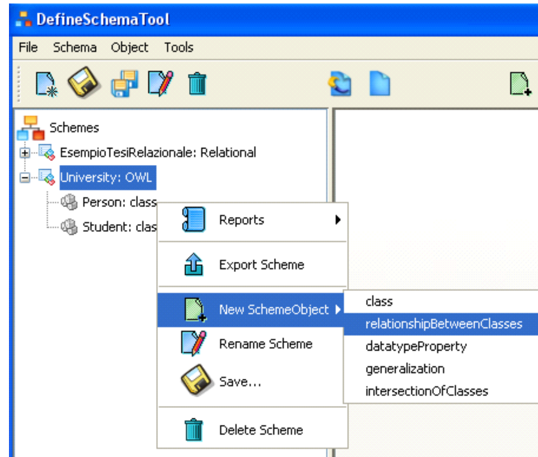


Figure 7.3: Specification of an OWL restriction through the RELATIONSHIPBETWEENCLASSES construct.

Data are handled in an object representation, where each construct is represented by a class. Then, according to the external system of interest, the persistence manager interacts with specific components. Therefore we add to the existing XML and IBM DB2 import/export tools three other tools able to import/export RDF, TopicMaps and OWL ontologies in RDF/XML syntax. Figure 7.4 shows some screenshots of the import/export tool.

After the schema definition phase, the user has to choose how to translate the schema. The designer has just to specify the source schema (and so its model) and the target model, and the system finds a translation.

Finally we can consider the activities of the metamodel engineer (MeE). *MeE* can define new basic transformations by writing Datalog rules or reusing some of the existing ones. The main task is the definition and management of the supermodel. This is a very delicate task and requires a good knowledge of data models as well as of the supermodel itself. Because of the nature of the supermodel, such tasks are quite infrequent: after a transitory phase where metaconstructs are introduced into the supermodel, translations and Skolem functions involving the new metaconstructs are created, modifications should tend to zero and reuse should be total.

Translation rules are stored in plain text files. A tool to support the user

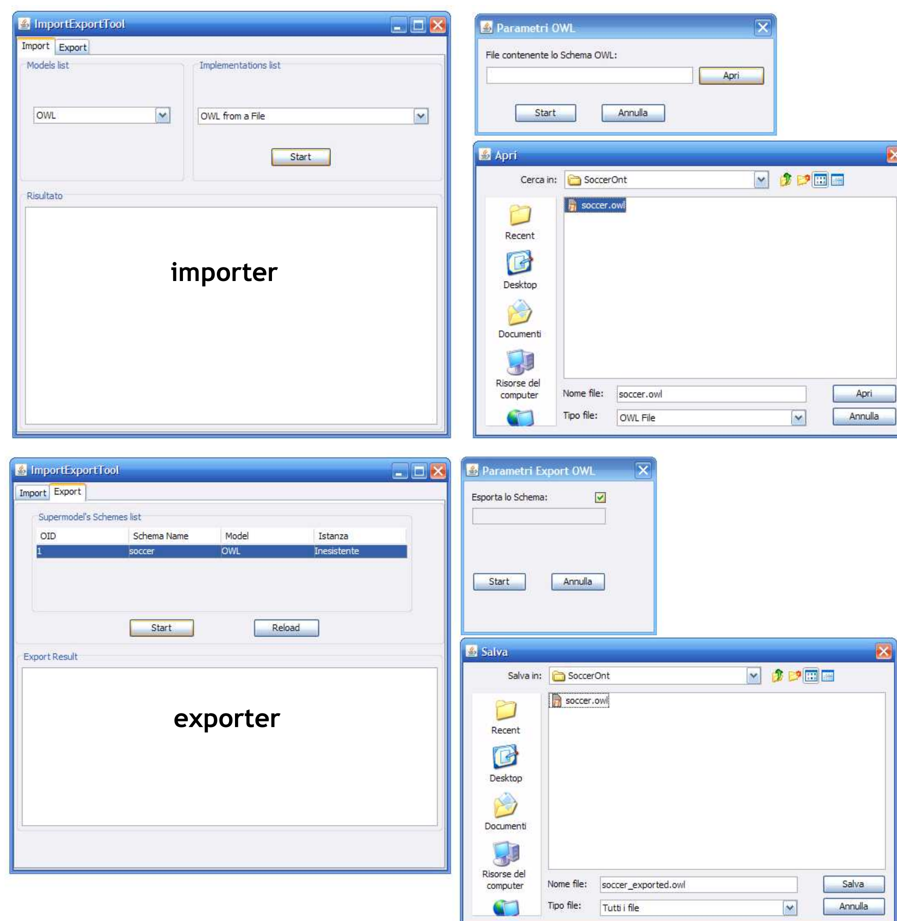


Figure 7.4: The import/export tool.

in the definition of translation rules has been developed. It supports Datalog syntax highlighting and auto-completion of literals, Skolem functions and variables used in the rule. These help functions are possible by leveraging on the metadata associated with the constructs and the Skolem functions stored in the repository. A Datalog translation rule uses Skolem functions. The tool provides a search feature to look up already defined Skolem functions for a specific construct, and allows the creation of new ones by selecting the target construct, giving the function a name and adding a number of parameters. Once a Skolem function is defined, its description is stored as metadata in the dictionary.

Actually we do not much care on the overall performance of the translation process, because this is not the focus of this work. However, it is worth mentioning that we decided to implement our own Datalog engine, because of the need for the OID invention feature and for the ease of integration with our relational dictionary. The algorithm that generates SQL statements from Datalog rules performs well. It generally takes seconds and it has a linear cost in the size of the input (i.e. number of rules). Performance of the translation executions depends on the number of SQL statements (number of rules) to be executed and on the number of join conditions each rule implies. Moreover, the structure of the dictionary and the materialization of Skolem functions do not help performance. In any case, even if efficiency can be improved, the translation of schemas is performed in a few seconds.

7.2 Experimental Results

To test all the features of the tool we mainly used synthetic schemas (for semantic annotations in various formats) and databases, in order to be cost-effective in the analysis of the various features of models and schemas.

We have tested the tool using two different points of views, one "in-the-small" and the other "in-the-large".

For the testing in-the-small, we performed two sets of experiments. The first set was driven by the rules: we tested every single Datalog rule of each basic translation, to verify the results of the individual substeps (An example of the set of rules wee use in our experiments is reported in the Appendices). The second set was driven by model features: we defined many (a few hundred) ad hoc schemas (for different kind of semantic annotations), each one with a specific pattern of constructs, in order to verify that such a pattern is handled as required. In this way, we have verified the results of basic translations.

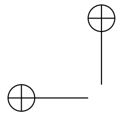
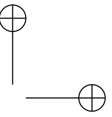
For the testing in-the-large, we used a set of more complex schemas. We considered some significant models, representatives of the various families of semantic annotations we have considered, and defined one schema for each of them, with all the features of such a model. Then, we translated them into other models of interest. In this case, the translation process for these schemas required the application of a number (from three to eight) of basic translations. Here, we initially built complex translations by manually composing basic ones (as this was the only way in the first version of our tool) and then experimented with the automatic generation, and obtained the same sequences of basic translations. In some cases, when there are various acceptable sequences, the tool generated one of them. A complete transformation example has been already presented in Chapter 6.6.

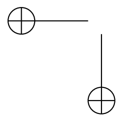
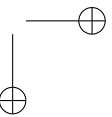
As stated before, due to the presence of many constraints, relationships and other "implicit" semantics, or even lacking, the translations are not always "complete". In a few cases the result of the translations can be manually edited by domain experts to improve the quality. However exploiting the metamodel approach we can provide convenient translations with minimum information loss, as demonstrated by our experiments.

7.3 Discussion

In this section we have presented the implementation of MIDST (Model Independent Data and Schema Translation), a framework for the development of an effective implementation of a generic (i.e., model independent) platform for schema and data translation. We have used this tool to perform a number (many hundred) of translations in the semantic annotation context.

Actually we do not much care on the overall performance of the translation process. However we are currently working to improve the tool and to make more efficient the overall translation process.

Conclusion

"Midway upon the road of our life I found myself within a dark wood, for the right way had been missed. Ah! how hard a thing it is to tell what this wild and rough and dense wood was, which in thought renews the fear! So bitter is it that death is little more."

Dante Alighieri *Inferno Canto I*

The Semantic Web is the new generation World Wide Web. It extends the Web by giving information a well defined meaning, allowing it to be processed by machines. This vision is going to become reality thanks to a set of technologies which have been specified and maintained by the World Wide Web Consortium (W3C), and more and more research efforts from the industry and the academia. Therefore, the basis for the Semantic Web are computer-understandable descriptions of resources. We can create such descriptions by annotating resources with metadata, resulting in "annotations" about that resource.

Semantic annotation is the creation of metadata and relations between them with the task of defining new methods of access to information and enriching the potentialities of the ones already existent.

Therefore, semantic annotations are used to enrich the informative content of Web documents and to express in more formal way the meaning of a resource. The goal is to create annotations with well-defined semantics, however those semantics may be defined.

A remarkable issue in the context of semantic annotations Remarkable importance is covered by semantic interoperability, because it introduces notable

challenges. The semantic interoperability is, in general, the ability to share the "meaning" of available information and of the applications built on them. From the semantic annotations point of view, this opens the possibility of operating with heterogenous resources by providing a bridge of common techniques and methods.

In this dissertation we focused on "model-generic" interoperability by means of translation of schemas and data. We discussed our recent results and our contributions to the development of the MIDST platform that allows the specification of the models of interest (mainly semantic annotation formalism and database), with all relevant details, and the generation of translations of their schemas from one model to another.

The usefulness of the MIDST proposal relies on the expressive power of its supermodel, that is the set of models handled and accuracy and precision of such models representation. In order to improve the expressive power of the supermodel, we extended it with more complex structured elements (such as collections) in order to properly represent semantic annotation data models.

We have exploited MIDST features in order to manage semantic annotation interoperability and integration extended the supermodel capabilities.

In practice, the contribute can be summarized as follows:

- we have introduced the concept of a general model to properly represent a broad range of data models. The proposed general model is based on the idea of *construct*: a construct represents a "structural" concept of a data model. We find out a construct for each "structural" concept of every considered data model and, hence, a data model can be completely represented by means of its constructs set.

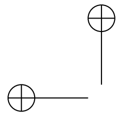
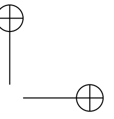
- we have extended the general model approach to properly represent all (ideally) Semantic Web data models with particular attention to semantic annotation models.

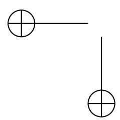
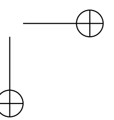
- we have devised a set of brand new Datalog-like rules to perform the translation between semantic annotation models an the integration of those models and data in databases.

- we implemented a flexible framework that allows to validate the concepts of the approach and to test their effectiveness.

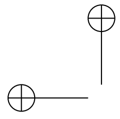
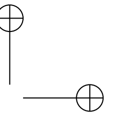
Currently, we are working on the following aspects:

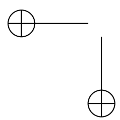
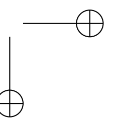
- the improvement of the rules system in order to augment the translation process;
- the re-engineering of the tool to improve the performance and the usability;
- the customization of translation, data level translations and applications of the technique to typical model management scenarios, such as schema evolution and ontology evolution

Appendices

A. Datalog Rules for RDB to OWL translations

In order to give an idea of how our Datalog rules (with OID invention) are defined, we report in this appendix the set of rules needed to translate relational database schemas into OWL ones. Due to space limitation we only report the most significative portion of this set.

We firstly report the macro rules and then specify the corpus of each basic rule.

Macrorules

1. Replace SM-AGGREGATIONs with ABSTRACTs.
2. Generalizations for SM-AGGREGATIONs which have SM-LEXICALs referencing only one table.
3. Replace SM-AGGREGATIONs with LEXICALs referencing two different tables with two SM-BINARYAGGREGATIONOFABSTRACTSs.
4. Replace SM-AGGREGATIONs which have only one foreign key with two SM-BINARYAGGREGATIONOFABSTRACTSs.
5. Replace SM-AGGREGATIONs which do not have only one foreign key with two SM-BINARYAGGREGATIONOFABSTRACTSs.
6. Creation of SM-ASSERTIONONPROPERTY to address inverse relations.
7. Management of restriction for SM-LEXICALs (Datatype properties) which are not foreign keys.

8. Management of restrictions for SM-Lexicals (Object properties) which are foreign keys.

Basic rules

The basic rules for each of the previous points are presented in the following:

1.

```
    SM_Abstract (
    OID: #AbstractOID_3*(oid),
    Name: name
)
<- SM_Aggregation (
    OID: oid,
    Name: name
), !TMP_AggWithTwoKeyReferencesWithoutOtherLexicals(
    OID: oid
) ;
```

2.

```
 SM_Generalization (
    OID: #GeneralizationOID_1*(aggToOID),
    Name: aggToName,
    ParentAbstractOID: #AbstractOID_3(aggToOID)
)

<-

TMP_AggWithSingleKeyReference(
    OID: aggOID
), SM_Lexical(
    OID: lexOID,
    isIdentifier: "true",
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
    LexicalFromOID: lexOID,
    ForeignKeyOID: fkOID
) , SM_ForeignKey(
    OID: fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
) , SM_Aggregation (
    OID: aggToOID,
    Name: aggToName
) ;

SM_ChildOfGeneralization (
    OID: #ChildOfGeneralizationOID_1*(aggOID),
    Name: aggName,
    GeneralizationOID: #GeneralizationOID_1(aggToOID),
    ChildAbstractOID: #AbstractOID_3(aggOID)
)

<-

TMP_AggWithSingleKeyReference(
    OID: aggOID
) , SM_Lexical(
    OID: lexOID,
    isIdentifier: "true",
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
```

```
    LexicalFromOID: lexOID,
    ForeignKeyOID: fkOID
) , SM_ForeignKey(
    OID: fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
) , SM_Aggregation (
    OID: aggOID,
    Name: aggName
) , SM_Aggregation (
    OID: aggToOID,
    Name: aggToName
) ;
```

3.

```
SM_BinaryAggregationOfAbstracts (
    OID: #BinaryAggregationOfAbstractsOID_6*(aggOID,agg1ToOID,agg2ToOID),
    Name: aggName+"_"+agg1ToName,
    isSymmetric: "false",
    isTransitive: "false",
    isDirected: "true",
    isOptional1: "false",
    isFunctional1: "false",
    isOptional2: "true",
    isFunctional2: "false",
    Abstract1OID: #AbstractOID_3(agg1ToOID),
    Abstract2OID: #AbstractOID_3(agg2ToOID)
)

<*-

SM_Aggregation (
    OID: aggOID,
    Name: aggName
) , TMP_AggWithTwoKeyReferencesWithoutOtherLexicals(
    OID: aggOID
) , SM_Lexical(
    OID: lex1OID,
    isIdentifier: "true",
    AggregationOID: aggOID
) , SM_Lexical(
    OID: lex2OID,
    isIdentifier: "true",
    AggregationOID: aggOID
), SM_ComponentOfForeignKey(
    LexicalFromOID: lex1OID,
    ForeignKeyOID: fk1OID
) , SM_ComponentOfForeignKey(
    LexicalFromOID: lex2OID,
    ForeignKeyOID: fk2OID
) , SM_ForeignKey(
    OID: fk1OID,
    AggregationFromOID: aggOID,
    AggregationToOID: agg1ToOID
) , SM_ForeignKey(
    OID: fk2OID,
    AggregationFromOID: aggOID,
    AggregationToOID: agg2ToOID
) , SM_Aggregation (
    OID: agg1ToOID,
    Name: agg1ToName
) , SM_Aggregation (
    OID: agg2ToOID,
    Name: agg2ToName
), agg1ToOID<>agg2ToOID ;
```

```
SM_AssertionOnProperty (
    OID: #AssertionOnPropertyOID_1*(aggOID,agg1ToOID,agg2ToOID),
    Name: binAggFromName+"_InverseOf_"+binAggToName,
    isInverse: "true",
    BinaryAggregationOfAbstracts1OID:
     #BinaryAggregationOfAbstractsOID_6(aggOID,agg1ToOID,agg2ToOID),
    BinaryAggregationOfAbstracts2OID:
     #BinaryAggregationOfAbstractsOID_6(aggOID,agg2ToOID,agg1ToOID)
)

<-

SM_Aggregation (
    OID: aggOID,
    Name: aggName
) ,

TMP_AggWithTwoKeyReferencesWithoutOtherLexicals(
    OID: aggOID
) , SM_Lexical(
    OID: lex1OID,
    isIdentifier: "true",
    AggregationOID: aggOID
) , SM_Lexical(
    OID: lex2OID,
    isIdentifier: "true",
    AggregationOID: aggOID
), SM_ComponentOfForeignKey(
    LexicalFromOID: lex1OID,
    ForeignKeyOID: fk1OID
) , SM_ComponentOfForeignKey(
    LexicalFromOID: lex2OID,
    ForeignKeyOID: fk2OID
) , SM_ForeignKey(
    OID: fk1OID,
    AggregationFromOID: aggOID,
    AggregationToOID: agg1ToOID
) , SM_ForeignKey(
    OID: fk2OID,
    AggregationFromOID: aggOID,
    AggregationToOID: agg2ToOID
) , SM_Aggregation (
    OID: agg1ToOID,
    Name: agg1ToName
) , SM_Aggregation (
    OID: agg2ToOID,
    Name: agg2ToName
), agg1ToOID<>agg2ToOID , SM_BinaryAggregationOfAbstracts [DEST] (
    OID: #BinaryAggregationOfAbstractsOID_6(aggOID,agg1ToOID,agg2ToOID),
    Name: binAggToName
) , SM_BinaryAggregationOfAbstracts [DEST] (
    OID: #BinaryAggregationOfAbstractsOID_6(aggOID,agg2ToOID,agg1ToOID),
    Name: binAggFromName
) ;
```

4.
```
SM_BinaryAggregationOfAbstracts(
        OID: #BinaryAggregationOfAbstractsOID_7*(lexOID,lex2OID),
        Name: lexName,
        isSymmetric: "false",
        isTransitive: "false",
        isDirected: "true",
        isOptional1: "true",
        isFunctional1: "true",
        isOptional2: "true",
```

```
    isFunctional2: "false",
    Abstract1OID: #AbstractOID_3(aggOID),
    Abstract2OID: #AbstractOID_3(aggToOID)
)

<-

SM_Aggregation (
    OID: aggOID
) , !TMP_AggWithSingleKeyReference (
    OID: aggOID
) , SM_Lexical(
    OID: lexOID,
    Name: lexName,
    isNullable: isnull,
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) , SM_ForeignKey(
    OID:fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
), SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggOID)
) , SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggToOID)
) ;


SM_BinaryAggregationOfAbstracts(
    OID: #BinaryAggregationOfAbstractsOID_7*(lex2OID,lexOID),
    Name: "InverseOf"+lexName,
    isSymmetric: "false",
    isTransitive: "false",
    isDirected: "true",
    isOptional1: "true",
    isFunctional1: "false",
    isOptional2: isnull,
    isFunctional2: "true",
    Abstract1OID: #AbstractOID_3(aggToOID),
    Abstract2OID: #AbstractOID_3(aggOID)
)

<-

SM_Aggregation (
    OID: aggOID
) , !TMP_AggWithSingleKeyReference (
    OID: aggOID
) , SM_Lexical(
    OID: lexOID,
    Name: lexName,
    isNullable: isnull,
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) , SM_ForeignKey(
    OID:fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
```

```
), SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggOID)
) , SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggToOID)
) ;
```

5.
```
SM_BinaryAggregationOfAbstracts(
    OID: #BinaryAggregationOfAbstractsOID_7*(lexOID,lex2OID),
    Name: lexName,
    isSymmetric: "false",
    isTransitive: "false",
    isDirected: "true",
    isOptional1: "true",
    isFunctional1: "true",
    isOptional2: "true",
    isFunctional2: "false",
    Abstract1OID: #AbstractOID_3(aggOID),
    Abstract2OID: #AbstractOID_3(aggToOID)
)

<-

TMP_AggWithSingleKeyReference (
    OID: aggOID
) , SM_Lexical(
    OID: lexOID,
    Name: lexName,
    isIdentifier: "false",
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) , SM_ForeignKey(
    OID:fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
) , SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggOID)
) , SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggToOID)
) ;


SM_BinaryAggregationOfAbstracts(
    OID: #BinaryAggregationOfAbstractsOID_7*(lex2OID,lexOID),
    Name: "InverseOf"+lexName,
    isSymmetric: "false",
    isTransitive: "false",
    isDirected: "true",
    isOptional1: "true",
    isFunctional1: "false",
    isOptional2: "true",
    isFunctional2: "true",
    Abstract1OID: #AbstractOID_3(aggToOID),
    Abstract2OID: #AbstractOID_3(aggOID)
)

<-

SM_Aggregation (
    OID: aggOID
) , TMP_AggWithSingleKeyReference (
    OID: aggOID
) , SM_Lexical(
```

```
    OID: lexOID,
    Name: lexName,
    isIdentifier: "false",
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) , SM_ForeignKey(
    OID:fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
), SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggOID)
) , SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggToOID)
) ;
```

6.

```
SM_AssertionOnProperty (
    OID: #AssertionOnPropertyOID_2*(lex1OID,lex2OID),
    Name: binAggFromName+"_InverseOf_"+binAggToName,
    isInverse: "true",
    BinaryAggregationOfAbstracts1OID: #BinaryAggregationOfAbstractsOID_7(lex2OID,lex1OID),
    BinaryAggregationOfAbstracts2OID: #BinaryAggregationOfAbstractsOID_7(lex1OID,lex2OID)
)

<-

SM_Aggregation (
    OID: aggOID
) , !TMP_AggWithSingleKeyReference (
    OID: aggOID
) , SM_Lexical(
    OID: lex1OID,
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lex1OID,
    LexicalToOID: lex2OID
) , SM_ForeignKey(
    OID:fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
), SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggOID)
) , SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggToOID)
) , SM_BinaryAggregationOfAbstracts [DEST] (
    OID: #BinaryAggregationOfAbstractsOID_7(lex2OID,lex1OID),
    Name: binAggFromName
) , SM_BinaryAggregationOfAbstracts [DEST] (
    OID: #BinaryAggregationOfAbstractsOID_7(lex1OID,lex2OID),
    Name: binAggToName
) ;

SM_AssertionOnProperty (
    OID: #AssertionOnPropertyOID_2*(lex1OID,lex2OID),
    Name: binAggFromName+"_InverseOf_"+binAggToName,
    isInverse: "true",
    BinaryAggregationOfAbstracts1OID: #BinaryAggregationOfAbstractsOID_7(lex2OID,lex1OID),
    BinaryAggregationOfAbstracts2OID: #BinaryAggregationOfAbstractsOID_7(lex1OID,lex2OID)
)

<-
```

```
SM_Aggregation (
    OID: aggOID
) , TMP_AggWithSingleKeyReference (
    OID: aggOID
) , SM_Lexical(
    OID: lex1OID,
    isIdentifier: "false",
    AggregationOID: aggOID
) , SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lex1OID,
    LexicalToOID: lex2OID
) , SM_ForeignKey(
    OID:fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
), SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggOID)
) , SM_Abstract [DEST] (
    OID: #AbstractOID_3(aggToOID)
) , SM_BinaryAggregationOfAbstracts [DEST] (
    OID: #BinaryAggregationOfAbstractsOID_7(lex2OID,lex1OID),
    Name: binAggFromName
) , SM_BinaryAggregationOfAbstracts [DEST] (
    OID: #BinaryAggregationOfAbstractsOID_7(lex1OID,lex2OID),
    Name: binAggToName
) ;
```

7.
```
SM_Lexical (
    OID: #LexicalOID_0*(lexOID),
    Name: lexName,
    isOptional: "true",
    isFunctional: "true",
    Type: type,
    AbstractOID: #AbstractOID_3(aggOID)
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    Type: type
) , !SM_ComponentOfForeignKey(
    LexicalFromOID: lexOID
) ;
```

8.
```
SM_Abstract (
    OID: #AbstractOID_5*(lexOID),
    Name: lexName+"_Restriction"
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), !SM_ComponentOfForeignKey(
```

```
    LexicalFromOID: lexOID
) ;


SM_Lexical (
    OID: #LexicalOID_9*(lexOID),
    Name: lexName+"_Restricted",
    isOptional: "false",
    isFunctional: "true",
    type: type,
    AbstractOID: absOID,
    AbstractAsLexicalOID: #AbstractOID_5(lexOID)
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Lexical[DEST](
    OID: #LexicalOID_0(lexOID),
    Name: lexName,
    isFunctional: isFunc,
    AbstractOID: absOID,
    type: type
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), !SM_ComponentOfForeignKey(
    LexicalFromOID: lexOID
) ;


SM_AssertionOnProperty (
    OID: #AssertionOnPropertyOID_3*(lexOID),
    Name: "MinCardRestrOn_"+lexName,
    isRestriction: "true",
    restrictionType: "MinCardinality",
    lexical1OID: #LexicalOID_9(lexOID),
    lexical2OID: #LexicalOID_0(lexOID)
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), !SM_ComponentOfForeignKey(
    LexicalFromOID: lexOID
) ;


SM_Generalization (
    OID: #GeneralizationOID_2*(lexOID),
    Name: "ChildOf_"+lexName+"_Restriction",
    ParentAbstractOID: #AbstractOID_5(lexOID)
)

<-

SM_Lexical(
```

```
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), !SM_ComponentOfForeignKey(
    LexicalFromOID: lexOID
) ;


SM_ChildOfGeneralization (
    OID: #ChildOfGeneralizationOID_2*(lexOID),
    Name: absName,
    ChildAbstractOID: #AbstractOID_3(aggOID),
    GeneralizationOID: #GeneralizationOID_2(lexOID)
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID),
    Name: absName
), !SM_ComponentOfForeignKey(
    LexicalFromOID: lexOID
) ;
```

8.
```
SM_Abstract (
    OID: #AbstractOID_5*(lexOID),
    Name: lexName+"_Restriction"
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), SM_BinaryAggregationOfAbstracts[DEST](
    OID: #BinaryAggregationOfAbstractsOID_7(lexOID,lex2OID)
), SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) ;


SM_BinaryAggregationOfAbstracts(
    OID: #BinaryAggregationOfAbstractsOID_8*(lexOID,lex2OID),
    Name: lexName+"_Restricted",
    isSymmetric: "false",
    isTransitive: "false",
    isDirected: "true",
    isOptional1: "false",
    isFunctional1: "true",
    isOptional2: "true",
    isFunctional2: "false",
```

```
    Abstract1OID: abs1OID,
    Abstract2OID: abs2OID,
    AbstractAsBinAggOfAbsOID: #AbstractOID_5(lexOID)
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_BinaryAggregationOfAbstracts[DEST](
    OID: #BinaryAggregationOfAbstractsOID_7(lexOID,lex2OID),
    Abstract1OID: abs1OID,
    Abstract2OID: abs2OID
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) , SM_ForeignKey(
    OID:fkOID,
    AggregationFromOID: aggOID,
    AggregationToOID: aggToOID
) ;


SM_AssertionOnProperty (
    OID: #AssertionOnPropertyOID_3*(lexOID),
    Name: "MinCardRestrOn_"+lexName,
    isRestriction: "true",
    restrictionType: "MinCardinality",
    binaryAggregationOfAbstracts1OID:
     #BinaryAggregationOfAbstractsOID_8(lexOID,lex2OID),
    binaryAggregationOfAbstracts2OID:
     #BinaryAggregationOfAbstractsOID_7(lexOID,lex2OID)
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_BinaryAggregationOfAbstracts[DEST](
    OID: #BinaryAggregationOfAbstractsOID_7(lexOID,lex2OID),
    Abstract1OID: abs1OID,
    Abstract2OID: abs2OID
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) ;


SM_Generalization (
    OID: #GeneralizationOID_2*(lexOID),
    Name: "ChildOf_"+lexName+"_Restriction",
    ParentAbstractOID: #AbstractOID_5(lexOID)
```

```
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID)
), SM_BinaryAggregationOfAbstracts[DEST](
    OID: #BinaryAggregationOfAbstractsOID_7(lexOID,lex2OID)
), SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) ;



SM_ChildOfGeneralization (
    OID: #ChildOfGeneralizationOID_2*(lexOID),
    Name: absName,
    ChildAbstractOID: #AbstractOID_3(aggOID),
    GeneralizationOID: #GeneralizationOID_2(lexOID)
)

<-

SM_Lexical(
    OID: lexOID,
    Name: lexName,
    AggregationOID: aggOID,
    isNullable: "false"
), SM_Abstract[DEST](
    OID: #AbstractOID_3(aggOID),
    Name: absName
), SM_BinaryAggregationOfAbstracts[DEST](
    OID: #BinaryAggregationOfAbstractsOID_7(lexOID,lex2OID)
), SM_ComponentOfForeignKey(
    OID: compFKOID,
    ForeignKeyOID: fkOID,
    LexicalFromOID: lexOID,
    LexicalToOID: lex2OID
) ;
```

B. Datalog Rules for OWL to RDB translations

In this appendix, we show the set of rules needed to translate OWL Semantic Annotation in relation databases. As for Appendix 7.3, due to space limitation, we only report the most significative portion of this set.

We firstly report the macro rules and then specify the corpus of each basic rule.

Macrorules

1. Preliminary selections
   a) Find named classes
   b) Find object property restrictions
   c) Find datatype property restrictions
   d) Find inverse object properties
2. Object properties hierarchy computation
   a) Object property hierarchy creation
   b) Find hierarchy roots
   c) Find object properties characteristics
3. Datatype properties hierarchy computation
   a) Datatype property hierarchy creation
   b) Find hierarchy roots
   c) Find Datatype properties characteristics

4. Class hierarchy computation
   a) Create parent-child relations
   b) Create class hierarchy
5. Selection of object properties that can be translated in the relational model
   a) Find object property owners
   b) Find object property characteristics
   c) Find the object properties that can be translated (with not NULL range)
6. Translation in relational model
   a) Translate object properties
   b) Translate datatype properties
   c) Translate generalizations

Basic Rules

The basic rules for each of the previous points are presented in the following:

1.

```
TMP_NamedClass [SOURCE](
    OID: oid
) <- SM_Abstract(
    OID: oid
) , !SM_BinaryAggregationOfAbstracts(
    AbstractAsBinAggOfAbsOID: oid
) , !SM_Lexical (
    abstractAsLexicalOID: oid
), !SM_Set(
    AbstractOID: oid,
    isAnonymous: "true"
) ;


TMP_ObjectPropertyRestriction [SOURCE] (
    PropOID: propOID,
    RestrPropOID: oid,
    RestrClassOID: absOID
) <- SM_BinaryAggregationOfAbstracts(
    OID: oid,
    AbstractAsBinAggOfAbsOID: absOID
), SM_Abstract(
    OID: absOID
), SM_AssertionOnProperty (
    isRestriction: "true",
    BinaryAggregationOfAbstracts1OID: oid,
    BinaryAggregationOfAbstracts2OID: propOID
```

```
) ;


TMP_DatatypePropertyRestriction [SOURCE] (
    PropOID: propOID,
    RestrPropOID: oid,
    RestrClassOID: absOID
) <- SM_Lexical(
    OID: oid,
    AbstractAsLexicalOID: absOID
), SM_Abstract(
    OID: absOID
), SM_AssertionOnProperty (
    isRestriction: "true",
    Lexical1OID: oid,
    Lexical2OID: propOID
) ;


TMP_ObjectPropertyInverse [SOURCE] (
    Prop1OID: prop1OID,
    Prop2OID: prop2OID
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: prop1OID
), SM_BinaryAggregationOfAbstracts (
    OID: prop2OID
), SM_AssertionOnProperty (
    BinaryAggregationOfAbstracts1OID: prop1OID,
    BinaryAggregationOfAbstracts2OID: prop2OID,
    isInverse: "true"
) ;


TMP_ObjectPropertyInverse [SOURCE] (
    Prop1OID: prop2OID,
    Prop2OID: prop1OID
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: prop1OID
), SM_BinaryAggregationOfAbstracts (
    OID: prop2OID
), SM_AssertionOnProperty (
    BinaryAggregationOfAbstracts1OID: prop1OID,
    BinaryAggregationOfAbstracts2OID: prop2OID,
    isInverse: "true"
), !TMP_ObjectPropertyInverse (
    Prop1OID: prop2OID,
    Prop2OID: prop1OID
) ;
```

2.

```
TMP_ObjectPropertyHierarchy [SOURCE] (
    ParentOID: parentOID,
    ChildOID: childOID
)

<-
```

```
SM_Generalization (
    OID: genOID,
    parentBinAggrOID: parentOID
), SM_ChildOfGeneralization (
    generalizationOID: genOID,
    childBinAggrOID: childOID
), SM_BinaryAggregationOfAbstracts (
    OID: parentOID,
    isEquivalence: "false"
), SM_BinaryAggregationOfAbstracts (
    OID: childOID,
    isEquivalence: "false"
) ;


TMP_ObjectPropertyRoot [SOURCE] (
    OID: oid
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: oid,
    isEquivalence: "false"
), !TMP_ObjectPropertyHierarchy (
    ChildOID: oid
), !TMP_ObjectPropertyRestriction (
    restrPropOID: oid
) ;


TMP_ObjectPropertyIsFunctional1 [SOURCE](
    PropOID: propOID,
    IsFunctional1: "true"
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: propOID,
    isFunctional1: "true",
    isEquivalence: "false"
), !TMP_ObjectPropertyRestriction (
    restrPropOID: propOID
);


TMP_ObjectPropertyIsFunctional2 [SOURCE](
    PropOID: propOID,
    IsFunctional2: "true"
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: propOID,
    isFunctional2: "true",
    isEquivalence: "false"
), !TMP_ObjectPropertyRestriction (
    restrPropOID: propOID
);


TMP_ObjectPropertyDomain [SOURCE](
    PropOID: propOID,
```

```
    Abstract1OID: domainOID
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: propOID,
    Abstract1OID: domainOID,
    isEquivalence: "false"
), SM_Abstract (
    OID: domainOID
), !TMP_ObjectPropertyRestriction (
    restrPropOID: propOID
);


TMP_ObjectPropertyRange [SOURCE](
    PropOID: propOID,
    Abstract2OID: rangeOID
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: propOID,
    Abstract2OID: rangeOID,
    isEquivalence: "false"
), SM_Abstract (
    OID: rangeOID
), !TMP_ObjectPropertyRestriction (
    restrPropOID: propOID
);


TMP_ObjectPropertyIsFunctional1 [SOURCE] (
    PropOID: childOID,
    isFunctional1: isFunc1
)

<-

TMP_ObjectPropertyIsFunctional1 (
    PropOID: parentOID,
    isFunctional1: isFunc1
), TMP_ObjectPropertyHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ObjectPropertyIsFunctional1 (
    PropOID: childOID
), SM_BinaryAggregationOfAbstracts (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: childOID
) ;



TMP_ObjectPropertyIsFunctional2 [SOURCE] (
    PropOID: childOID,
    isFunctional2: isFunc2
)

<-

TMP_ObjectPropertyIsFunctional2 (
    PropOID: parentOID,
```

```
    isFunctional2: isFunc2
), TMP_ObjectPropertyHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ObjectPropertyIsFunctional2 (
    PropOID: childOID
), SM_BinaryAggregationOfAbstracts (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: childOID
) ;


TMP_ObjectPropertyDomain [SOURCE] (
    PropOID: childOID,
    Abstract1OID: domainOID
)

<-

TMP_ObjectPropertyDomain (
    PropOID: parentOID,
    Abstract1OID: domainOID
), TMP_ObjectPropertyHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ObjectPropertyDomain (
    PropOID: childOID
), SM_BinaryAggregationOfAbstracts (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: childOID
) ;


TMP_ObjectPropertyRange [SOURCE] (
    PropOID: childOID,
    Abstract2OID: rangeOID
)

<-

TMP_ObjectPropertyRange (
    PropOID: parentOID,
    Abstract2OID: rangeOID
), TMP_ObjectPropertyHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ObjectPropertyRange (
    PropOID: childOID
), SM_BinaryAggregationOfAbstracts (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: childOID
) ;
```

3.

```
TMP_DatatypePropertyHierarchy [SOURCE] (
    ParentOID: parentOID,
    ChildOID: childOID
)

<-

SM_Generalization (
```

```
    OID: genOID,
    parentLexicalOID: parentOID
), SM_ChildOfGeneralization (
    generalizationOID: genOID,
    childLexicalOID: childOID
), SM_Lexical (
    OID: parentOID
), SM_Lexical (
    OID: childOID
), !TMP_DatatypePropertyRestriction (
    restrPropOID: parentOID
), !TMP_DatatypePropertyRestriction (
    restrPropOID: childOID
) ;


TMP_DatatypePropertyRoot [SOURCE] (
    OID: oid
)

<-

SM_Lexical (
    OID: oid
), !TMP_ObjectPropertyHierarchy (
    ChildOID: oid
), !TMP_DatatypePropertyRestriction (
    restrPropOID: oid
) ;


TMP_DatatypePropertyIsFunctional [SOURCE] (
    PropOID: propOID,
    IsFunctional: "true"
)

<-

SM_Lexical (
    OID: propOID,
    IsFunctional: "true"
), !TMP_DatatypePropertyRestriction (
    restrPropOID: propOID
) ;


TMP_DatatypePropertyRange [SOURCE] (
    PropOID: propOID,
    Type: type
)

<-

SM_Lexical (
    OID: propOID,
    Type: type
), !TMP_DatatypePropertyRestriction (
    restrPropOID: propOID
), type<>"'null'" ;


TMP_DatatypePropertyDomain [SOURCE] (
    PropOID: propOID,
    AbstractOID: domainOID
)
```

```
<-

SM_Lexical (
    OID: propOID,
    AbstractOID: domainOID
), !TMP_DatatypePropertyRestriction (
    restrPropOID: propOID
), SM_Abstract(
  OID: domainOID
) ;


TMP_DatatypePropertyIsFunctional [SOURCE] (
    PropOID: propOID,
    IsFunctional: isFunc
)

<-

TMP_DatatypePropertyRoot(
    OID: propOID
), SM_Lexical (
    OID: propOID,
    IsFunctional: isFunc
), !TMP_DatatypePropertyIsFunctional (
    PropOID: propOID
) ;


TMP_DatatypePropertyRange [SOURCE] (
    PropOID: propOID,
    Type: type
)

<-

TMP_DatatypePropertyRoot (
    OID: propOID
), SM_Lexical (
    OID: propOID,
    Type: type
), !TMP_DatatypePropertyRange (
    PropOID: propOID
) ;


TMP_DatatypePropertyDomain [SOURCE] (
    PropOID: propOID,
    AbstractOID: domainOID
)

<-

TMP_DatatypePropertyRoot (
  OID: propOID
), SM_Lexical (
    OID: propOID,
    AbstractOID: domainOID
), !TMP_DatatypePropertyDomain (
    PropOID: propOID
) ;


TMP_DatatypePropertyIsFunctional [SOURCE] (
    PropOID: childOID,
    isFunctional: isFunc
```

```
    )

    <-

    TMP_DatatypePropertyIsFunctional (
        PropOID: parentOID,
        isFunctional: isFunc
    ), TMP_DatatypePropertyHierarchy (
        ParentOID: parentOID,
        ChildOID: childOID
    ), !TMP_DatatypePropertyIsFunctional (
        PropOID: childOID
    ), SM_Lexical (
        OID: parentOID
    ), SM_Lexical (
        OID: childOID
    ) ;


    TMP_DatatypePropertyRange [SOURCE] (
        PropOID: childOID,
        Type: type
    )

    <-

    TMP_DatatypePropertyRange (
        PropOID: parentOID,
        Type: type
    ), TMP_DatatypePropertyHierarchy (
        ParentOID: parentOID,
        ChildOID: childOID
    ), !TMP_DatatypePropertyRange (
        PropOID: childOID
    ), SM_Lexical (
        OID: parentOID
    ), SM_Lexical (
        OID: childOID
    ) ;


    TMP_DatatypePropertyDomain [SOURCE] (
        PropOID: childOID,
        AbstractOID: domainOID
    )

    <-

    TMP_DatatypePropertyDomain (
        PropOID: parentOID,
        AbstractOID: domainOID
    ), TMP_DatatypePropertyHierarchy (
        ParentOID: parentOID,
        ChildOID: childOID
    ), !TMP_DatatypePropertyDomain (
        PropOID: childOID
    ), SM_Lexical (
        OID: parentOID
    ), SM_Lexical (
        OID: childOID
    ) ;
```

4.
```
TMP_ParentChildHierarchy [SOURCE] (
        ParentOID: parentOID,
        ChildOID: childOID
```

```
)

<-

TMP_NamedClass (
    OID: parentOID
), TMP_NamedClass (
    OID: childOID
), SM_Generalization (
    OID: genOID,
    parentAbstractOID: parentOID
), SM_ChildOfGeneralization (
    OID: childGenOID,
    GeneralizationOID: genOID,
    ChildAbstractOID: childOID
), SM_Abstract (
    OID: parentOID
), SM_Abstract (
    OID: childOID
) ;


TMP_ParentChildHierarchy [SOURCE] (
    ParentOID: parentOID,
    ChildOID: childOID
)

<-

TMP_NamedClass (
    OID: childOID
), TMP_NamedClass (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    Abstract1OID: childOID,
    Abstract2OID: intersOID,
    isEquivalence: "true"
), SM_Set (
    OID: setOID,
    AbstractOID: intersOID
), SM_ComponentOfSet (
    SetOID: setOID,
    AbstractOID: parentOID
), SM_Abstract (
    OID: parentOID
), SM_Abstract (
    OID: childOID
), !TMP_ParentChildHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
) ;



TMP_ParentChildHierarchy [SOURCE] (
    ParentOID: parentOID,
    ChildOID: childOID
)

<-

TMP_NamedClass (
    OID: childOID
), TMP_NamedClass (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
```

```
    Abstract1OID: childOID,
    Abstract2OID: intersOID,
    isEquivalence: "true"
), SM_Set (
    OID: setOID,
    AbstractOID: intersOID
), SM_ComponentOfSet (
    SetOID: setOID,
    AbstractOID: parentOID
), SM_Abstract (
    OID: parentOID
), SM_Abstract (
    OID: childOID
), SM_Generalization (
    OID: genOID,
    parentAbstractOID: intersOID
), SM_ChildOfGeneralization (
    OID: childGenOID,
    GeneralizationOID: genOID,
    ChildAbstractOID: childOID
), !TMP_ParentChildHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
) ;



TMP_ExtendedClassHierarchy [SOURCE] (
    ParentOID: parentOID,
    ChildOID: childOID
) <- TMP_ParentChildHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), SM_Abstract (
    OID: childOID
), SM_Abstract (
    OID: parentOID
) ;


TMP_ExtendedClassHierarchy [SOURCE] (
    ParentOID: grandparentOID,
    ChildOID: childOID
)

<*-

TMP_ParentChildHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), TMP_ParentChildHierarchy (
    ParentOID: grandparentOID,
    ChildOID: parentOID
), !TMP_ExtendedClassHierarchy (
    ParentOID: grandparentOID,
    ChildOID: childOID
), SM_Abstract (
    OID: childOID
), SM_Abstract (
    OID: parentOID
), SM_Abstract (
    OID: grandparentOID
) ;


TMP_ClassAncestor [SOURCE](
```

```
    ChildOID: childOID,
    AncestorOID: ancestorOID
)

<*-

TMP_ExtendedClassHierarchy (
    ChildOID: childOID,
    ParentOID: ancestorOID
), TMP_ExtendedClassHierarchy (
    ChildOID: childOID,
    ParentOID: parentOID
), TMP_ExtendedClassHierarchy (
    ChildOID: parentOID,
    ParentOID: ancestorOID
), SM_Abstract (
    OID: childOID
), SM_Abstract (
    OID: parentOID
), SM_Abstract (
    OID: ancestorOID
) ;


TMP_DirectClassHierarchy [SOURCE] (
    ChildOID: childOID,
    ParentOID: parentOID
)

<-

TMP_ParentChildHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ClassAncestor (
    ChildOID: childOID,
    AncestorOID: parentOID
), SM_Abstract (
    OID: childOID
), SM_Abstract (
    OID: parentOID
) ;
```

5.
```
TMP_ObjectPropertyRestrictionOwner [SOURCE] (
    ClassOID: oid,
    PropOID: propOID,
    RestrPropOID: restrProp1OID
)

<-

SM_Abstract (
    OID: oid
), TMP_NamedClass (
    OID: oid
), SM_Abstract (
    OID: intersOID
), SM_BinaryAggregationOfAbstracts (
    Abstract1OID: oid,
    Abstract2OID: intersOID,
    isEquivalence: "true"
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrProp1OID
), SM_Set (
```

```
    OID: setOID,
    isAnonymous: "true",
    AbstractOID: intersOID
), SM_ComponentOfSet (
    SetOID: setOID,
    AbstractOID: comp1OID
), TMP_ObjectPropertyRestriction (
    PropOID: propOID,
    RestrClassOID: comp1OID,
    RestrPropOID: restrProp1OID
) ;


TMP_ObjectPropertyRestrictionOwner [SOURCE] (
    ClassOID: oid,
    PropOID: propOID,
    RestrPropOID: restrProp1OID
)

<-

SM_Abstract (
    OID: oid
), TMP_NamedClass (
    OID: oid
), SM_Abstract (
    OID: intersOID
), SM_Generalization (
    OID: genOID,
    ParentAbstractOID: intersOID
), SM_ChildOfGeneralization (
    generalizationOID: genOID,
    childAbstractOID: oid
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrProp1OID
), SM_Set (
    OID: setOID,
    isAnonymous: "true",
    AbstractOID: intersOID
), SM_ComponentOfSet (
    SetOID: setOID,
    AbstractOID: comp1OID
), TMP_ObjectPropertyRestriction (
    PropOID: propOID,
    RestrClassOID: comp1OID,
    RestrPropOID: restrProp1OID
) ;



TMP_ObjectPropertyRestrictionOwner [SOURCE] (
    ClassOID: oid,
    PropOID: propOID,
    RestrPropOID: restrPropOID
)

<-

SM_Abstract (
    OID: oid
), TMP_NamedClass (
    OID: oid
), SM_Abstract (
    OID: restrClassOID
```

```
), SM_BinaryAggregationOfAbstracts (
    Abstract1OID: oid,
    Abstract2OID: restrClassOID,
    isEquivalence: "true"
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrPropOID
), TMP_ObjectPropertyRestriction (
    PropOID: propOID,
    RestrClassOID: restrClassOID,
    RestrPropOID: restrPropOID
) ;


TMP_ObjectPropertyRestrictionOwner [SOURCE] (
    ClassOID: oid,
    PropOID: propOID,
    RestrPropOID: restrPropOID
)

<-

SM_Abstract (
    OID: oid
), TMP_NamedClass (
    OID: oid
), SM_Abstract (
    OID: restrClassOID
), SM_Generalization (
    OID: genOID,
    ParentAbstractOID: restrClassOID
), SM_ChildOfGeneralization (
    generalizationOID: genOID,
    childAbstractOID: oid
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrPropOID
), TMP_ObjectPropertyRestriction (
    PropOID: propOID,
    RestrClassOID: restrClassOID,
    RestrPropOID: restrPropOID
) ;



TMP_ObjectPropertyRestrictionOwner [SOURCE] (
    ClassOID: oid,
    PropOID: propOID,
    RestrPropOID: restrProp1OID
)

<-

SM_Abstract (
    OID: oid
), TMP_NamedClass (
    OID: intersOID
), SM_Abstract (
    OID: intersOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrProp1OID
), SM_Set (
```

```
    OID: setOID,
    isAnonymous: "false",
    AbstractOID: intersOID
), SM_ComponentOfSet (
    SetOID: setOID,
    AbstractOID: comp1OID
), TMP_ObjectPropertyRestriction (
    PropOID: propOID,
    RestrClassOID: comp1OID,
    RestrPropOID: restrProp1OID
) ;


TMP_OBJECTPROPERTYOWNER [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID
)

<*-

SM_Abstract (
    OID: classOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), TMP_ObjectPropertyRestrictionOwner (
    PropOID: propOID,
    ClassOID: classOID
) ;


TMP_OBJECTPROPERTYOWNER [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID
)

<-

SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyDomain (
    PropOID: propOID,
    Abstract1OID: classOID
), !TMP_ObjectPropertyRestriction(
    RestrPropOID: propOID
), !TMP_OBJECTPROPERTYOWNER (
    PropOID: propOID,
    ClassOID: classOID
) ;



TMP_OBJECTPROPERTYOWNER [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID
)

<-

SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyDomain (
    PropOID: propOID,
    Abstract1OID: domainOID
), TMP_NamedClass(
    OID: classOID
), !TMP_OBJECTPROPERTYOWNER (
```

```
    PropOID: propOID,
    ClassOID: classOID
), !SM_Abstract (
    OID: domainOID
), !TMP_ObjectPropertyRestriction (
    restrPropOID: propOID
), !TMP_ParentChildHierarchy (
    ChildOID: classOID
) ;


TMP_ObjectPropertyOwnerWithParentOwner [SOURCE] (
    ClassOID: classOID,
    PropOID: propOID
)

<-

TMP_ObjectPropertyOwner (
    ClassOID: classOID,
    PropOID: propOID
), TMP_ParentChildHierarchy (
    ParentOID: parentOID,
    ChildOID: classOID
), TMP_ObjectPropertyOwner (
    ClassOID: parentOID,
    PropOID: propOID
), SM_Abstract (
    OID: classOID
), SM_Abstract (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
) ;


TMP_ObjectPropertyRootOwner [SOURCE] (
    ClassOID: classOID,
    PropOID: propOID
) <-

TMP_ObjectPropertyOwner (
    ClassOID: classOID,
    PropOID: propOID
), !TMP_ObjectPropertyOwnerWithParentOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_Abstract (
    OID: classOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
) ;


TMP_ObjectPropertyRange4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    Abstract2OID: abs2OID
)

<-

TMP_ObjectPropertyOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_AssertionOnProperty (
```

```
    BinaryAggregationOfAbstracts1OID: restrPropOID,
    BinaryAggregationOfAbstracts2OID: propOID,
    isRestriction: "true",
    restrictionType: "AllValuesFrom"
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyRestrictionOwner (
    ClassOID: classOID,
    PropOID: propOID,
    RestrPropOID: restrPropOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrPropOID,
    Abstract2OID: abs2OID
), SM_Abstract (
    OID: abs2OID
) ;


TMP_ObjectPropertyIsOptional4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    isOptional1: isOpt
)

<-

TMP_ObjectPropertyOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_AssertionOnProperty (
    BinaryAggregationOfAbstracts1OID: restrPropOID,
    BinaryAggregationOfAbstracts2OID: propOID,
    isRestriction: "true",
    restrictionType: "MinCardinality"
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyRestrictionOwner (
    ClassOID: classOID,
    PropOID: propOID,
    RestrPropOID: restrPropOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrPropOID,
    isOptional1: isOpt
) ;


TMP_ObjectPropertyIsFunctional4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    isFunctional1: isFunc
)

<-

TMP_ObjectPropertyOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_AssertionOnProperty (
    BinaryAggregationOfAbstracts1OID: restrPropOID,
    BinaryAggregationOfAbstracts2OID: propOID,
    isRestriction: "true",
    restrictionType: "MaxCardinality"
```

```
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyRestrictionOwner (
    ClassOID: classOID,
    PropOID: propOID,
    RestrPropOID: restrPropOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrPropOID,
    isFunctional1: isFunc
) ;


TMP_ObjectPropertyIsOptional4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    isOptional1: isOpt
)

<-

TMP_ObjectPropertyOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_AssertionOnProperty (
    BinaryAggregationOfAbstracts1OID: restrPropOID,
    BinaryAggregationOfAbstracts2OID: propOID,
    isRestriction: "true",
    restrictionType: "Cardinality"
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyRestrictionOwner (
    ClassOID: classOID,
    PropOID: propOID,
    RestrPropOID: restrPropOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrPropOID,
    isOptional1: isOpt
) ;



TMP_ObjectPropertyIsFunctional4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    isFunctional1: isFunc
)

<-

TMP_ObjectPropertyOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_AssertionOnProperty (
    BinaryAggregationOfAbstracts1OID: restrPropOID,
    BinaryAggregationOfAbstracts2OID: propOID,
    isRestriction: "true",
    restrictionType: "Cardinality"
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyRestrictionOwner (
    ClassOID: classOID,
    PropOID: propOID,
```

```
    RestrPropOID: restrPropOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: restrPropOID,
    isFunctional1: isFunc
) ;



TMP_ObjectPropertyRange4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    Abstract2OID: rangeOID
)

<*-

TMP_ObjectPropertyRootOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), TMP_ObjectPropertyRange (
    PropOID: propOID,
    Abstract2OID: abs2OID
), SM_Abstract (
    OID: classOID
), !SM_Abstract (
    OID: abs2OID
), !TMP_ObjectPropertyRange4Owner (
    PropOID: propOID,
    ClassOID: classOID
), SM_Abstract (
    OID: rangeOID
), TMP_NamedClass (
    OID: rangeOID
), !TMP_ParentChildHierarchy (
    ChildOID: rangeOID
) ;



TMP_ObjectPropertyRange4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    Abstract2OID: abs2OID
)

<*-

TMP_ObjectPropertyRootOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), TMP_ObjectPropertyRange (
    PropOID: propOID,
    Abstract2OID: abs2OID
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyOwner (
    PropOID: propOID,
    ClassOID: classOID
), !TMP_ObjectPropertyRange4Owner (
    PropOID: propOID,
```

```
    ClassOID: classOID
) ;


TMP_ObjectPropertyIsOptional4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    isOptional1: isOpt
)

<*-

TMP_ObjectPropertyRootOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID,
    isOptional1: isOpt,
    isEquivalence: "false"
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyOwner (
    PropOID: propOID,
    ClassOID: classOID
), !TMP_ObjectPropertyIsOptional4Owner (
    PropOID: propOID,
    ClassOID: classOID
) ;


TMP_ObjectPropertyIsFunctional4Owner [SOURCE] (
    PropOID: propOID,
    ClassOID: classOID,
    isFunctional1: isFunc
)

<*-

TMP_ObjectPropertyRootOwner (
    ClassOID: classOID,
    PropOID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), TMP_ObjectPropertyIsFunctional1 (
    PropOID: propOID,
    isFunctional1: isFunc
), SM_Abstract (
    OID: classOID
), TMP_ObjectPropertyOwner (
    PropOID: propOID,
    ClassOID: classOID
), !TMP_ObjectPropertyIsFunctional4Owner (
    PropOID: propOID,
    ClassOID: classOID
) ;


TMP_ObjectPropertyRange4Owner [SOURCE] (
    ClassOID: childOID,
    PropOID: propOID,
    Abstract2OID: abs2OID
)

<-

TMP_ObjectPropertyRange4Owner (
```

```
    ClassOID: parentOID,
    PropOID: propOID,
    Abstract2OID: abs2OID
), TMP_DirectClassHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ObjectPropertyRange4Owner (
    ClassOID: childOID,
    PropOID: propOID
), SM_Abstract (
    OID: childOID
), SM_Abstract (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
) ;


TMP_ObjectPropertyIsOptional4Owner [SOURCE] (
    ClassOID: childOID,
    PropOID: propOID,
    isOptional1: isOpt
)

<-

TMP_ObjectPropertyIsOptional4Owner (
    ClassOID: parentOID,
    PropOID: propOID,
    isOptional1: isOpt
), TMP_DirectClassHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ObjectPropertyIsOptional4Owner (
    ClassOID: childOID,
    PropOID: propOID
), SM_Abstract (
    OID: childOID
), SM_Abstract (
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
) ;


TMP_ObjectPropertyIsFunctional4Owner [SOURCE] (
    ClassOID: childOID,
    PropOID: propOID,
    isFunctional1: isFunc
)

<-

TMP_ObjectPropertyIsFunctional4Owner (
    ClassOID: parentOID,
    PropOID: propOID,
    isFunctional1: isFunc
), TMP_DirectClassHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), !TMP_ObjectPropertyIsFunctional4Owner (
    ClassOID: childOID,
    PropOID: propOID
), SM_Abstract (
    OID: childOID
), SM_Abstract (
```

```
    OID: parentOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
) ;


TMP_ObjectPropertyTransformableNoConsider2 [SOURCE] (
    DomainOID: domain1OID,
    PropOID: prop1OID,
    RangeOID: range1OID
)

<-

SM_BinaryAggregationOfAbstracts (
    OID: prop1OID
), SM_BinaryAggregationOfAbstracts (
    OID: prop2OID
), SM_Abstract (
    OID: domain1OID
), SM_Abstract (
    OID: domain2OID
), SM_Abstract (
    OID:range1OID
), SM_Abstract (
    OID: range2OID
), TMP_ObjectPropertyTransformable (
    DomainOID: domain1OID,
    PropOID: prop1OID,
    RangeOID: range1OID
), TMP_ObjectPropertyTransformable (
    DomainOID: domain2OID,
    PropOID: prop2OID,
    RangeOID: range2OID
), !TMP_ObjectPropertyTransformableNoConsider (
    DomainOID: domain1OID,
    PropOID: prop1OID,
    RangeOID: range1OID
), !TMP_ObjectPropertyTransformableNoConsider (
    DomainOID: domain2OID,
    PropOID: prop2OID,
    RangeOID: range2OID
), SM_AssertionOnProperty (
    BinaryAggregationOfAbstracts1OID: prop1OID,
    BinaryAggregationOfAbstracts2OID: prop2OID,
    isInverse: "true"
), domain1OID = range2OID, domain2OID = range1OID ;



TMP_ObjectPropertyToTransform [SOURCE] (
    DomainOID: domain1OID,
    PropOID: prop1OID,
    RangeOID: range1OID

)

<-

TMP_ObjectPropertyTransformable (
    DomainOID: domain1OID,
    PropOID: prop1OID,
    RangeOID: range1OID
), !TMP_ObjectPropertyTransformableNoConsider (
    DomainOID: domain1OID,
    PropOID: prop1OID,
```

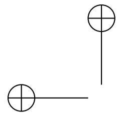
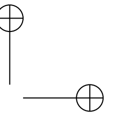
```
    RangeOID: range1OID
), !TMP_ObjectPropertyTransformableNoConsider2 (
    DomainOID: domain1OID,
    PropOID: prop1OID,
    RangeOID: range1OID
), SM_Abstract (
    OID: domain1OID
), SM_Abstract (
    OID: range1OID
), SM_BinaryAggregationOfAbstracts (
    OID: prop1OID
) ;


TMP_ManyToManyRelationship [SOURCE] (
    DomainOID: domainOID,
    PropOID: propOID,
    RangeOID: rangeOID
)

<-

SM_Abstract (
    OID: domainOID
), SM_Abstract (
    OID: rangeOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), TMP_ObjectPropertyIsFunctional2 (
    PropOID: propOID,
    isFunctional2: "false"
), TMP_ObjectPropertyToTransform (
    DomainOID: domainOID,
    PropOID: propOID,
    RangeOID: rangeOID
), !TMP_ObjectPropertyInverse (
    Prop1OID: propOID
), TMP_ObjectPropertyIsFunctional4Owner (
    ClassOID: domainOID,
    PropOID: propOID,
    isFunctional1: "false"
) ;


TMP_ManyToManyRelationship [SOURCE] (
    DomainOID: domainOID,
    PropOID: propOID,
    RangeOID: rangeOID
)

<-

SM_Abstract (
    OID: domainOID
), SM_Abstract (
    OID: rangeOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID
), SM_BinaryAggregationOfAbstracts (
    OID: propInvOID
), TMP_ObjectPropertyToTransform (
    DomainOID: domainOID,
    PropOID: propOID,
    RangeOID: rangeOID
), TMP_ObjectPropertyInverse (
    Prop1OID: propOID,
```

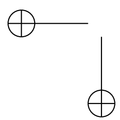
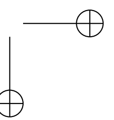
```
    Prop2OID: propInvOID
), TMP_ObjectPropertyIsFunctional4Owner (
    ClassOID: domainOID,
    PropOID: propOID,
    isFunctional1: "false"
), TMP_ObjectPropertyIsFunctional2 (
    PropOID: propOID,
    isFunctional2: "false"
), TMP_ObjectPropertyIsFunctional4Owner (
    ClassOID: rangeOID,
    PropOID: propInvOID,
    isFunctional1: "false"
), TMP_ObjectPropertyIsFunctional2 (
    PropOID: propInvOID,
    isFunctional2: "false"
) ;
```

6.

```
SM_Lexical (
    OID: #Lexical4BinAggOfAbsAndDomainAbsAndRangeAbsAndRoleOID_4
     *(propOID,domainOID,rangeOID,role1),
    Name: "InverseOf"+propName,
    IsIdentifier: "false",
    IsNullable: isOpt,
    type: type,
    AggregationOID: #Aggregation4AbsOID_1(domainOID)
)

<-

SM_Abstract (
    OID: domainOID
), SM_Abstract (
    OID: rangeOID
), SM_BinaryAggregationOfAbstracts (
    OID: propOID,
    Name: propName,
    Role1: role1
), TMP_OneToOneOrManyRelationship (
    DomainOID: domainOID,
    PropOID: propOID,
    RangeOID: rangeOID,
    isDirect: "false"
), TMP_ObjectPropertyIsOptional4Owner (
    ClassOID: domainOID,
    PropOID: propOID,
    isOptional1: isOpt
), SM_Lexical [DEST] (
    OID: #Lexical4AbsWithoutKeyOID_1(rangeOID),
    Type: type
) ;


SM_ForeignKey (
    OID: #ForeignKey4DomainAbsAndRangeAbsAndBinAggrOfAbsAndRoleOID_4
    *(domainOID,rangeOID,propOID,role1),
    Name: aggName+"_To_"+domainName,
    AggregationFromOID:
     #Aggregation4AbsAndAbsAndBinAggOfAbsOID_3(domainOID,rangeOID,propOID),
    AggregationToOID: #Aggregation4AbsOID_1(domainOID)
)

<-

TMP_ManyToManyRelationship (
    DomainOID: domainOID,
```

```
    RangeOID: rangeOID,
    PropOID: propOID
), SM_Abstract (
    OID: domainOID,
    Name: domainName
), SM_Abstract (
    OID: rangeOID,
    Name: rangeName
), SM_BinaryAggregationOfAbstracts (
    OID: propOID,
    Name: propName,
    Role1: role1
), SM_Aggregation [DEST] (
    OID: #Aggregation4AbsAndAbsAndBinAggOfAbsOID_3
      (domainOID,rangeOID,propOID),
    Name: aggName
) ;


SM_Lexical (
    OID: #Lexical4LexAndAbsOID3_2*(propOID,domainOID),
    Name: "Value",
    isIdentifier: "true",
    isNullable: "false",
    type: type,
    AggregationOID: #Aggregation4AbsAndLexOID_2(domainOID,propOID)
)

<-

SM_Abstract (
    OID: domainOID
), SM_Lexical (
    OID: propOID
), TMP_DatatypePropertyMultiValued (
    DomainOID: domainOID,
    PropOID: propOID
), TMP_DatatypePropertyRange4Owner (
    PropOID: propOID,
    ClassOID: domainOID,
    Type: type
) ;



SM_ForeignKey (
    OID: #ForeignKey4AbsParentAndAbsChild_2*(parentOID,childOID),
    Name: name2+"_isChildOf_"+name1,
    AggregationFromOID: #Aggregation4AbsOID_1(childOID),
    AggregationToOID: #Aggregation4AbsOID_1(parentOID)
)

<-

TMP_ParentChildHierarchy (
    ParentOID: parentOID,
    ChildOID: childOID
), SM_Abstract (
    OID: parentOID,
    Name: name1
), SM_Abstract (
    OID: childOID,
    Name: name2
) ;
```

Bibliography

[ACB05] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. A multilevel dictionary for model management. In *ER*, pages 160–175, 2005.

[ACB06] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Model-independent schema and data translation. In *EDBT*, pages 368–385, 2006.

[ACT+08] Paolo Atzeni, Paolo Cappellari, Riccardo Torlone, Philip A. Bernstein, and Giorgio Gianforme. Model-independent schema translation. *VLDB J.*, 17(6):1347–1370, 2008.

[AGC08] Paolo Atzeni, Giorgio Gianforme, and Paolo Cappellari. Reasoning on data models in schema translation. In Sven Hartmann and Gabriele Kern-Isberner, editors, *FoIKS*, volume 4932 of *Lecture Notes in Computer Science*, pages 158–177. Springer, 2008.

[AT93] Paolo Atzeni and Riccardo Torlone. A metamodel approach for the management of multiple models and translation of schemes. *Inf. Syst.*, 18(6):349–362, 1993.

[AT96] Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. pages 79–95. Springer, 1996.

[BC02] Richard V. Benjamins and Jesús Contreras. Six challenges for the semantic web. 2002.

[BCG+02] Sean Bechhofer, Les Carr, Carole A. Goble, Simon Kampa, and Timothy Miles-Board. The semantics of semantic annotation. In *On the Move to Meaningful Internet Systems, 2002 -*

DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002, pages 1152–1167, London, UK, 2002. Springer-Verlag.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.

[BG03] Petra S. Bayerl and Ulrike Gut. Methodology for reliable schema development and evaluation of manual annotations. In *Proceedings of the Workshop on Knowledge Markup and Semantic Annotation at the Second International Conference on Knowledge Capture (K-CAP 2003*, 2003.

[BGMN08] Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing mapping composition. *VLDB J.*, 17(2):333–353, 2008.

[BH07] Philip A. Bernstein and Howard Ho. Model management and schema mappings: Theory and practice. In *VLDB*, pages 1439–1440, 2007.

[BHJ+00] Philip A. Bernstein, Laura M. Haas, Matthias Jarke, Erhard Rahm, and Gio Wiederhold. Panel: Is generic metadata management feasible? In *VLDB*, pages 660–662, 2000.

[BHP00] Philip A. Bernstein, Alon Y. Halevy, and Rachel Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[BHS04] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. In *Handbook on Ontologies*, pages 3–28. 2004.

[BL99] Tim Berners-Lee. *Weaving the Web.* Texere Publishing Ltd., November 1999.

[BLFM05] Tim Berners-Lee, R. Fielding, and L. Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic syntax. http://tools.ietf.org/html/rfc3986, 2005.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[BM07] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2007. ACM.

[CGY07] Nadine Cullot, Raji Ghawi, and Kokou Ytongnon. Db2owl : A tool for automatic database-to-ontology mapping. In Michelangelo Ceci, Donato Malerba, and Letizia Tanca, editors, *SEBD*, pages 491–494, 2007.

[CHL+04] Kevin Chen-Chuan Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured databases on the web: observations and implications. *SIGMOD Rec.*, 33(3):61–70, 2004.

[CW03] Fabio Ciravegna and Yorick Wilks. Designing adaptive information extraction for the semantic web in amilcare. In *Annotation for the Semantic Web, Frontiers in Artificial Intelligence and Applications. IOS.* Press, 2003.

[DCES04] Souripriya Das, Eugene Inseok Chong, George Eadon, and Jaannathan Srinivasan. Supporting ontology-based semantic matching in rdbms. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1054–1065. VLDB Endowment, 2004.

[DH05] Anhai Doan and Alon Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26:83–94, 2005.

[dLC05] Cristian Pérez de Laborda and Stefan Conrad. Relational.owl: a data and schema representation format based on owl. In *APCCM '05: Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*, pages 89–96, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[FKPT05] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.

[Gar05] Lars Marius Garshol. A model for topic maps: Unifying rdf and topic maps. In *Extreme Markup Languages*, 2005.

[GM05] Lars Marius Garshol and Graham Moore. Iso/iec 13250-2: Topic maps data model. http://www.isotopicmaps.org/sam/sam-model/, 2005.

[GNP] Alexander Gruenstein, John Niekrasz, and Matthew Purver. Meeting structure annotation - annotations collected with a general purpose toolkit. In *Recent Trends in Discourse and Dialogue*, volume 39 of *Text, Speech and Language Technology*, pages 247–271.

[Gru95] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43(5-6):907–928, 1995.

[Hab07] Benjamin Habegger. Mapping a database into an ontology : an interactive relational learning approach. In *ICDE*, pages 1443–1447, 2007.

[Han05] Siegfried Handschuh. *Creating Ontology-based Metadata by Annotation for the Semantic Web*. PhD thesis, University of Karlsruhe (TH), 2005.

[HK87] Richard Hull and Roger King. Semantic database modeling: survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.

[HSC02] Siegfried Handschuh, Steffen Staab, and Fabio Ciravegna. S-cream - semi-automatic creation of metadata. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 358–372, London, UK, 2002. Springer-Verlag.

[HSV03] Siegfried Handschuh, Steffen Staab, and Raphael Volz. On deep annotation. In *WWW*, pages 431–438, 2003.

[HY90] Richard Hull and Masatoshi Yoshikawa. Ilog: declarative creation and manipulation of object identifiers. In *Proceedings of the sixteenth international conference on Very large databases*, pages 455–468, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[JSUJ06] J. Carletta, S. Evert, U. Heid, and J. Kilgour. The nite xml toolkit: data model and query language. *Language Resources and Evaluation Journal*, June 2006. AMI-1.

[KGHP] Aditya Kalyanpur, Jennifer Golbeck, James Hendler, and Bijan Parsia. Representation formalisms and methods .

[KKPS02] José Kahan, Marja-Riitta Koivunen, Eric Prud'hommeaux, and Ralph R. Swick. Annotea: an open rdf infrastructure for shared web annotations. *Computer Networks*, 39(5):589–608, 2002.

[Kog01a] Paul Kogut. Aerodaml: Applying information extraction to generate daml annotations from web pages. In *First International Conference on Knowledge Capture (K-CAP 2001). Workshop on Knowledge Markup and Semantic Annotation*, 2001.

[Kog01b] Paul Kogut. Aerodaml: Applying information extraction to generate daml annotations from web pages. In *First International Conference on Knowledge Capture (K-CAP 2001). Workshop on Knowledge Markup and Semantic Annotation*, 2001.

[Kri06] Madhav Krishna. Retaining semantics in relational databases by mapping them to rdf. In *WI-IATW '06: Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology*, pages 303–306, Washington, DC, USA, 2006. IEEE Computer Society.

[KVBF07] Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Internet Computing*, 11(6):60–67, 2007.

[Lau08] Georg Lausen. Relational databases in rdf: Keys and foreign keys. pages 43–56, 2008.

[Len02] Maurizio Lenzerini. Data integration: a theoretical perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, New York, NY, USA, 2002. ACM.

[MBR01] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *VLDB*, pages 49–58, 2001.

[MDCG03] Enrico Motta, John Domingue, Liliana Cabral, and Mauro Gaspari. Irs-ii: A framework and infrastructure for semantic web services. pages 306–318. Springer-Verlag, 2003.

[Mel04] Sergey Melnik. *Generic Model Management: Concepts And Algorithms (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[MH03] Jayant Madhavan and Alon Y. Halevy. Composing mappings among data sources. In *VLDB*, pages 572–583, 2003.

[MHS07] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between owl and relational databases. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 807–816, New York, NY, USA, 2007. ACM.

[MKH03] Kamil Matousek, Zdenek Kouba, and Petr Husták. Resource annotation and outline creation tool (rat-o). In *DEXA Workshops*, pages 80–83, 2003.

[MS01] Alexander Maedche and Steffen Staab. Ontology learning for the semantic web. *IEEE Intelligent Systems*, 16(2):72–79, 2001.

[MTP06] Alexander Mikroyannidis, Babis Theodoulidis, and Andreas Persidis. Parmenides: Towards business intelligence discovery from web data. In *WI '06: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 1057–1060, Washington, DC, USA, 2006. IEEE Computer Society.

[NBM07] Alan Nash, Philip A. Bernstein, and Sergey Melnik. Composition of mappings given by embedded dependencies. *ACM Trans. Database Syst.*, 32(1):4, 2007.

[NFM00] N.F. Noy, R.W. Fergerson, and M.A. Musen. The knowledge model of protégé-2000: Combining interoperability and flexibility. *Lecture Notes in Computer Science*, 1937:69–82, 2000.

[NG06] John Niekrasz and Alexander Gruenstein. NOMOS: A semantic web software framework for annotation of multimodal corpora. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, Genoa, Italy, May 2006.

[NSD+01] Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, and Mark A. Musen. Creating semantic web contents with protégé-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

[OMG05] Omg model driven architecture. Internet document, `http://www.omg.org/mda/`, 2005.

[OMG06] Mof: Omg's metaobject facility. Internet document, `http://www.omg.org/mof/`, 2006.

[PB03] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 826–873, 2003.

[POSV04] Abhijit A. Patil, Swapna A. Oundhakar, Amit P. Sheth, and Kunal Verma. Meteor-s web service annotation framework. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 553–562, New York, NY, USA, 2004. ACM.

[PVM+02] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating web data. In *VLDB*, pages 598–609, 2002.

[RDH+04] Alan Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. Owl pizzas: Practical experience of teaching owl-dl: Common errors & common patterns. pages 63–81. 2004.

[RH05] Lawrence Reeve and Hyoil Han. Survey of semantic annotation platforms. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1634–1638, New York, NY, USA, 2005. ACM.

[SE05] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. pages 146–171. 2005.

[SGR08] Amit P. Sheth, Karthik Gomadam, and Ajith Ranabahu. Semantics enhanced services: Meteor-s, sawsdl and sa-rest. *IEEE Data Eng. Bull.*, 31(3):8–12, 2008.

[SHN07] Ronald Schroeter, Jane Hunter, and Andrew Newman. Annotating relationships between multiple mixed-media digital objects by extending annotea. In *ESWC*, pages 533–548, 2007.

[SHZZ06] Guohua Shen, Zhiqiu Huang, Xiaodong Zhu, and Xiaofei Zhao. Research on the rules of mapping from relational model to owl. In Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace, editors, *Proceedings of the OWLED '06 Workshop on OWL: Experiences and Directions*, pages 21–29, 2006.

[SP05] Peyman Sazedj and H. Sofia Pinto. Time to evaluate: Targeting annotation tools, November 2005.

[TBA06] Quang Trinh, Ken Barker, and Reda Alhajj. Rdb2ont: A tool for generating owl ontologies from relational database systems. In *AICT-ICIW '06: Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, page 170, Washington, DC, USA, 2006. IEEE Computer Society.

[Top01] TopicMaps.org. Xml topic maps (xtm) 1.0. Technical report, TopicMaps.org, 2001.

[UW97] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. 1997.

[VHS+04] Raphael Volz, Siegfried Handschuh, Steffen Staab, Ljiljana Stojanovic, and Nenad Stojanovic. Unveiling the hidden bride: deep annotation for mapping and migrating legacy data to the semantic web. *Journal of Web Semantics*, 1(2):187–206, 2004.

[VvMD+02a] Maria Vargas-vera, Enrico Motta, John Domingue, Mattia Lanzoni, and Fabio Ciravegna. Mnm: Ontology driven semi-automatic and automatic support for semantic markup. pages 379–391. Springer Verlag, 2002.

[VVMD+02b] Maria Vargas-Vera, Enrico Motta, John Domingue, Mattia Lanzoni, Arthur Stutt, and Fabio Ciravegna. Mnm: Ontology driven semi-automatic and automatic support for semantic markup. In *EKAW*, pages 379–391, 2002.

[XCDS04] Zhuoming Xu, Xiao Cao, Yisheng Dong, and Wenping Su. Formal approach and automated tool for translating er schemata into owl ontologies. In *PAKDD*, pages 464–475, 2004.

[YHGS03] Yeliz Yesilada, Simon Harper, Carole Goble, and Robert Stevens. Ontology based semantic annotation for enhancing mobility support for visually impaired web users. In *In K-CAP 2003 Workshop on Knowledge Markup and Semantic Annotation*, 2003.