UNIVERSITY OF LATVIA
FACULTY OF COMPUTING

Guntars Būmans

Relational Database information availability to Semantic Web technologies

DOCTORAL THESIS
FOR DR.SC.COMP ACADEMIC DEGREE

FIELD:	COMPUTER SCIENCE
SUB-FIELD:	PROGRAMMING LANGUAGES AND SYSTEMS
SCIENTIFIC ADVISOR:	DR.SC.COMP. PROF. KĀRLIS ČERĀNS

RIGA, 2012

This work has been supported by the European Social Fund within the project «Support for Doctoral Studies at University of Latvia».

Acknowledgements

I would like to thank my supervisor Kārlis Čerāns, who helped me during all time of carrying out the thesis, at the beginning helping to find research direction, after that not allowing me to idle for too long, but with valuable ideas and inducements making me to continue the work. He did not spare his time to help in helping to solve the research problems, and involved in technical and organizatorial matters.

Thanks to all the anonymous reviewers of the publications for their constructive criticism, which has been very useful for further studies and contributed to the progress of the work of the thesis.Many thanks also to my family, especially my wife Anita, for the understanding, support and with feeling. Thank you for bearing the deficit of my time and attention!

Contents

1 Introduction

This work is concerned with ensuring the data availability for the semantic layer of the World Wide Web and with use of semantic technologies in data integration on both, the World Wide Web scale and on the enterprise level. In particular, we are interested in connecting the relational database data to the semantic technology landscape in the context of semantic re-engineering of existing relational data sets.

The main problem looked at in the thesis is a design of high level declarative mapping specification language between relational database and OWL formats. We study the existing approaches to the problem and define a new mapping language RDB2OWL for generating RDF triples corresponding to OWL ontology from relational database data, with stress on mapping readability by humans and having high level language constructs suitable for concise handling of practical use cases. Implementation of RDB2OWL language is described and done for subset of the language constructs. As a case study, we have specified in RDB2OWL language the mapping between 6 databases of the Latvian Medicine registries and corresponding ontologies.

1.1 Semantic Web and Semantic Technologies

Semantic Web [1] contains a group of methods, technologies and tools to make the huge information in the World Wide Web processable by machines and also semantically "understandable" by machines. Among others, the major standards serving this purpose are RDF [2], RDFS [3] and OWL [4, 5].

RDF is a language to describe information on resources in the World Wide Web. The information is described as a list of statements each statement being Subject-Predicate-Object triple. Subject- resource on which the statement is made, Predicate- some property of the subject and Object is value of the property for the subject. Subject and Predicate must be resources (URI), Object can be either resource or literal value. Subject of any statement is resource and as such it can be another statement. Semantically it means that one statement is about another statement. Parts of one triple can reference (use the same URIs) the resources of another triple, actually creating links between triples. From the data structure point of view RDF triple sets are oriented graphs with Subject and Object parts denoting nodes and Predicates standing for arcs directed from Subject to Object nodes.

As an example, consider a simple statement "Population of Riga the capital of Latvia is 706413". RDF graph (shown below) holds this information and uses fictious vocabulary http://geography. The graph image is generated from RDF Validation Service [6]:

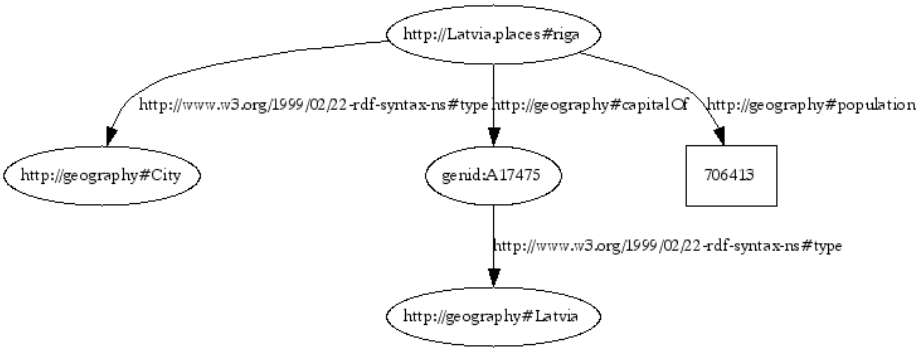


Figure 1. Sample RDF graph containing information about population of the capital of Latvia

References to blank nodes (genid:A17475 in the example) are made only from within enclosing the RDF graph. In the example *http://Latvia.places#riga* is capital of something and this something is of type *http://geography#Latvia*. This something is of no meaning outside the graph.

Triple set expressing the same information as in the sentence:

Subject	Predicate	Object
<http://Latvia.places#riga>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://geography#City>
<http://Latvia.places#riga>	<http://geography#capitalOf>	<genid:A17475>
<genid:A17475>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://geography#Latvia>
<http://Latvia.places#riga>	<http://geography#population>	"706413"

The same expressed in RDF/XML serialization:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:geography="http://geography#">

  <geography:City rdf:about="http://Latvia.places#riga">
    <geography:capitalOf>
      <geography:Latvia/>
    </geography:capitalOf>
    <geography:population> 706413 </geography:population>
  </geography:City>
</rdf:RDF>
```

RDFS (RDF schema) [3] is a language which allows for semantic extension of RDF by classes and properties that groups resources and defines associations between them. Classes and properties are the main. RDF document can use RDFS vocabularies to specify types of resources and properties (e.g. *http://geography#City*, http://geography#capitalOf). Web Ontology Language OWL [4] extends RDFS capabilities even further by introducing more features on classes and properties (cardinalities, property types: functional, inverse functional, transitive, inverse of, class expressions etc).

OWL as a knowledge representation language has semantic of the SROIQ description logic (a subset of first order logic and computable) ([4] sect. 2.3). There are reasoning tools applicable to OWL ontologies, e.g. Pellet [7] and FaCT++ [8]. The OWL class instances (individuals) and property values can be serialized in RDF

triples and exposed globally to the World Wibe Web. Therefore, the Semantic Web standards RDF, RDFS, OWL and other with help of supporting tools enable information to be coded in globally accessible, machine processable way, and also more conforming to semantics and logic of domain of discourse than table based information coding systems (e.g. relational tables).

1.2 The need for semantic re-engineering of Relational Databases

Development of Semantic Web has been effected for more than a decade. As a result, there are many semantic tools for semantic data management but few data in semantic form (e.g. RDF triple stores) because most of them still reside in relational databases, closed for public access. Therefore, it is a need of high importance to migrate or publish data from relational databases to RDF triple stores. The development community and organizations have responded to this need. The W3C SWEO Linking Open Data community project [9] is about extending the Web with data by publishing open data sets as RDF on the Web and by establishing RDF links between data items from different data sources to enable navigation. The project homepage [9] reports "Collectively, the 295 data sets consist of over 31 billion RDF triples, which are interlinked by around 504 million RDF links (September 2011)". A notable part of these published data has come from relational databases but more are still to come. By using some mapping specification techniques, is possible publishing the relational data as RDF triple set that corresponds to the target ontology.

The possibility to define RDB-to-RDF/OWL mappings efficiently has emerged as an issue of primary importance also in Semantic Latvia approach [10] and its application to the practical semantic re-engineering of medical domain data in Latvia [11, 12]. This approach proposes to create ontology (ontologies) for data that are available in a specific domain (e.g. government or medical data) using visual graphical notation offered by OWLGrEd [13,14] or UML/OWL profile [15], followed by RDB data integration into the format of the defined conceptual ontology, then followed by providing tools that are able to access the semantic data by means of a visual SPARQL query endpoint [11,16].

Organizations and governments from many countries have agreed on Transparency and Open Government Memorandum. For example, the US Open Government Directive of December 8, 2009 [17] demands that all agencies should publish at least three high-value data sets online and register them on data.gov. The UK government promotes raw data publishing as RDF on the Web. In UK public data website launched by Tim Berners-Lee [18], everybody can browse the published open linked data, use SPARQL endpoints or search engine to query them. Taking into consideration that relational databases schemas differ from ontologies and vocabularies for RDF to which data should be published the task can be time consuming. A reasonable approach is to publish raw data as they are in databases. For example portal data.gov.uk gives opportunity to access (browse or query by SPARQL) some raw data as RDF. People can build applications over those raw data or write code to transform them into meaningful ontologies. In this scenario, the burden of re-engineering of relational databases is left on open development

community. This is in accordance to what Tim Berner-Lee said on talk in TED2009 conference [19] "Raw Data Now". Re-engineering thus has two phases: in the first one, the data publisher technically publishes the raw data, and in the other phase, the open community makes data transformation into various domain ontologies. In order to do these tasks more effectively we need methods and tools for semantic re-engineering.

1.3 RDB-to-RDF/OWL mapping solutions

The task of RDB-to-RDF/OWL mapping is, given a relational database data, enable access to those in form of RDF triples <subject, predicate, object> corresponding to certain target RDF schema or OWL ontology. The RDF/OWL access to RDB data may be implemented either in form of transforming the RDB data into RDF format, or by providing a view mechanism for direct access of original RDB data in appropriate RDF format. Depending from the concrete mapping task, the target ontology or RDF schema may or may not be specified prior to the mapping specification.

There is a basic understanding about mapping relational table records to corresponding RDFS/OWL class instances, table columns to RDFS/OWL data properties and relations between tables (e.g. foreign keys) as RDFS/OWL object properties, however, different mapping formalisms differ on the concrete means that are available for the mapping specification.

Some of the most notable approaches dealing with RDB to RDF/OWL data mapping are Relational.OWL [20, 21], R2O [22], D2RQ [23], Virtuoso RDF Views [24] and DartGrid [25], Ultrawrap [26], Triplify [27]. There is W3C RDB2RDF Working Group [28] related to standardization of RDB to RDF mappings, as well as a related published survey of mapping RDBs to RDF [29].

We distinguish two mapping types between RDB and RDF/OWL. We call mapping a "**direct mapping**" if it defines a mapping from data in a relational database to RDF Graph representation with structure and vocabulary (ontology) directly corresponding to schema of the database. In short: triples from mapped RDF are instances of ontology with classes and attributes resembling design of database tables and columns. The direct mappings can help in publishing to the World Wide Web the data residing in relational databases in the form of RDF triples but resources are short to make the appropriate restructuring of data. After exposing the raw data in RDF format, any one is free to write appropriate processing code and thus promoting an open development.

The other type of mapping called a "**mapping language"** defines customized mappings from RDB to RDF datasets expressed in structure and vocabulary of author's choice. In typical cases, triples in RDF datasets conform to some preexistent ontology independently from database. We say that the *source database* is mapped to the *target ontology* (thought as conceptual model for the database).

Relational.OWL [20, 21] is a direct mapping solution that enables transformation of relational schema and data to OWL and RDF coding and therefore to be accessible from SPARQL endpoint. Thus, relational data are transformed to data according to the technical ontology (in one-to-one relation to the relational schema). It gives base

for further mapping by expressing correspondences between the technical and target OWL ontologies in SPARQL language.

R2O [22] platform consists of declarative mapping language between relational DB and OWL ontology and of tools (Mapster) to process these mappings. R2O language requires rather lengthy writing if done by hand, but some help for this is in user interface of Mapster. R2O is suitable for automatic mapping code generation.

D2RQ [23] is platform consisting of mapping language between RDB and OWL ontology and tools (D2R server) to process these mappings to enable SPARQL execution using relational data as virtual RDF graphs. The D2RQ mapping language is RDF based (typically written in n3 format) is easier to write and more readable than R2O, supports any SQL expressions usage (not the case with R2O).

Virtuoso RDF Views [24] also has language to express mapping between RDB and OWL ontologies and has tools to process SPARQL accessing relational data on the fly through these mappings. Virtuoso RDF RDB-to-OWL mapping language is more complicated than R2O and D2RQ thus giving option to specify more execution details (e.g., functions for URI patterns).

Dartgrid [25] is a toolset for semantic web to enable RDB-to-RDF/OWL mapping definition and query information from relational databases by SPARQL language. There is no special mapping language but mappings are stored in tables. Visual tools help for mapping editing. Dartgrid toolset supports application development that integrate existing relational dartabases for semantic information quering, searching (also full-text) and navigation.

Ultrawrap [26] and **Triplify** [27] platforms use SQL elaboration to enable triple generation from RDB data. The drawback is in need to write manually the SQL commands for triple generation. Triplify has also small plug-in (PHP) which added to web application root enables triplifying of relational data when triplifying SQLs and configuration (connections) are provided.

Currently there is new candidate W3C Standard (February 2012) for **a Direct Mapping of Relational Data to RDF** [30]. It states how to transform the data of relational databases into form of RDF graphs whose element structure and dictionary directly resembles element structure and names of the relational schema. The direct mapping Standard describes how to form RDF triples from database table rows and relations between tables, from primary and foreign keys (also multiple column keys), etc. The obtained data that correspond to the "technical" data schema can be afterwards transformed into a conceptual one either by means of SPARQL Construct queries, as in Relational.OWL [20, 21] approach, or by means of some RDF-to-RDF mapping language such as R2R [33], or some model transformation language (see e.g. [36] for an example approach). The Standard provides a uniform way for publishing the raw relational data into RDF. Up to now, there have been several different ways for this, for example, Relational.OWL and Ultrawrap approaches.

Currently there is also a new candidate W3C Standard (February 2012) mapping language **R2RML** [31], whose aim is to express custom mappings between RDB to RDF datasets where source and target structures may be different. R2RML language can establish correspondence from the source database to target (domain) ontology. R2RML mapping constructs specifies how to form subject, predicate and object parts of RDF triples by using information about the source relational schema structure (tables, columns, relations) and values in other defined constructs. R2RML language

source code is RDF document in Turtle syntax [32]. The language is rather low level and technically SQL oriented. R2RML as a Standard does not mean that other mapping languages do not make sense anymore; they have their specific syntactic and semantic advantages. A meaningful approach would be to provide compilers from other mapping languages to R2RML and then it would be possible to use R2RML supporting tools, for example, for triple generation.

Spyder [34] is a new RDB-RDF/OWL mapping language solution. It has its own native mapping language (Revelytix RDB Mapping language) and in addition, the R2RML Standard language is supported for triple generation. The native language is designed with similar ideas as of D2RQ but has additional properties the D2RQ is missing: 1) formal description of the source database schema and target ontology; 2) maximal use of references; 3) lessen code repetitions by using references to defined constructs and URI formats; 4) usage of implicit information to shorten expressions; etc.

The initial RDB-to-RDF mapping definition by means of hand-coded SQL statements as outlined in [38] has appeared less than satisfactory in practice, as did the approach of hand-coding the mappings in a low-level model transformation language over the intermediate data representation forms in a MOF-based repository.

1.4 Main Results

We offer a soundly motivated and practically efficient approach for RDB-to-RDF/OWL mapping that is suitable to cope with the motivating practical examples, as well as, being extensible beyond those. Our solution contains a mapping language and implementation framework briefly described below.

We propose a high level, human readable and machine processable declarative RDB-to-RDF/OWL mapping specification language RDB2OWL based on re-using the target ontology structure as a backbone where mapping expressions are written in the form of annotations to ontology classes and properties, as well as to the ontology itself. The RDB2OWL mapping specification language allows keeping the mapping definition fully human-comprehensible also in the case of complex mapping structure. It has simple MOF-style mapping metamodel (that can be re-phrased easily also into a mapping OWL ontology). Some RDB2OWL language features are:

- reuse of RDB table column and key information, whenever that is available;
- concrete human readable syntax for mapping expressions that is very simple and intuitive in the simple cases, and can also handle more advanced cases;
- built-in and user defined scalar and aggregate functions (including column-valued functions); function definition expressions can include references to source and auxiliary database tables and columns to enhance expressiveness;
- advanced mapping definition primitives, e.g. multiclass conceptualization that avoids the need of specifying long filtering conditions arising due to fixing a missing conceptual structure on large database tables;
- a possibility to resort to auxiliary structures defined on SQL level (e.g. user defined permanent and temporary tables, as well as SQL views), still maintaining the principle that the source RDB is to be kept read only.

The user readability and attachment of mapping expressions as annotations to ontology classes and properties allows the use of RDB2OWL language also as documentation means in describing the mappings from conceptual model onto the database design model.

Technically, we base the RDB2OWL execution environment on a designated relational database schema for mapping information storage, from which a two-phase SQL processing generates RDF triples corresponding to the target ontology. The first phase SQL execution processes mapping information to generate SQL sentences for triple creation from source database and the second phase execute the SQL scripts generated in the first phase. This approach benefits from SQL processing speed of modern RDBMS, combining a high-level specification language with efficient implementation structure.

On a practical side, we report on the experience of building RDB-to-OWL mapping for real life case of six Latvian medical registries [11], [12] within the presented simple mapping specification structure.

The RDF triple generation on the basis of an intermediate RDB-to-RDF mapping encoding within a relational database schema (the RDB2OWL mapping DB schema) has been successfully implemented by re-engineering the Latvian Medical registry databases (42,8 million triples have been generated in 20 minutes from mappings stored in special RDB schema). On the other hand, we annotated the Latvian Medical registry ontology in our RDB2OWL mapping language. Implementation of full set of RDB2OWL constructs is in progress including syntax level parsing, syntax model transformation to semantic model and to the intermediate execution model (RDB2OWL mapping DB schema).

The novelty of our approach is:

- human readability and conciseness of RDB2OWL mapping expressions due to high level mapping constructs and reuse of source database and target ontology structure;
- mapping pattern observations from real life examples that is supported by RDB2OWL mapping language (e.g. multi-class conceptualization);
- execution architecture with mappings stored in relational database and two phase SQL execution.

2 RDB-to-RDF/OWL mapping task

2.1 RDB and semantic format comparison

There are substantial differences comparing relational database model [35] of RDB vs. RDFS/OWL data models:

- relational database schemas are design models for implementation purposes but OWL ontologies typically are created as conceptual models;
- naming of entities and relations in relational databases are technically oriented whereas conceptual names from specific domain are used in RDF/OWL;
- relational databases are not aware of subclass relation but it is one of the most basic relation used in ontologies;
- it is not possible to have n:n relation between two database tables (there is need for the third table) but properties based on many to many relation in RDFS/OWL are natural;
- relations (foreign key) between database tables do not have means to express qualifier constraints, stating, for example, that number of linked rows is between 1 and 2 but RDF/OWL can naturally express cardinality constraints on domain or range classes. Databases can easily express 0..1 cardinality by using the NOT NULL constraint. Program code in table triggers can implement also the other constraints but such coding practice actually buries the qualifier information deep into the source code and hides from the model.

2.2 The need for RDB-to-RDF/OWL mapping solutions

The recent years are characterized by increasing use of semantic technologies both on a global scale (Semantic Web) and locally within enterprises, supported by the development of open definitions and standards such as RDF [1], SPARQL 1.1 [51], OWL [4] and many others. The number and performance of tools for semantic content management has grown and continues to grow. However, majority of data continue to reside in relational databases. Some reasons: relational databases are efficient in terms of processing time and data volume, they have precise definition, many SQL based tools and application development environments, many existing applications use data in relational databases.

In this situation, there is a need of high importance for efficient information integration between “the old world” of relational databases and “the new information world” with semantic standards and supporting tools. For this purpose, the research and technology development has been aimed at bridging relational databases (RDB) to RDF/OWL by mapping languages and techniques. The starting point was paper “Relational Databases on the Semantic Web” by T.Berners-Lee [39] back in 1998 and then followed a number of successful approaches: R_2O [22], D2RQ [23], Virtuoso

RDF Views [24], DartGrid [25] as well as on UltraWrap [26],Triplify [27] and Spyder [34] and others.

Most of these approaches are concentrating on efficient machine processing of the mappings, often preferably querying RDBs on the fly from an SPARQL-enabled endpoint. Much less attention, however, has been given to creating high-level mapping definitions that are oriented towards readability for a human being and that have a capacity to handle complex database-to-ontology/RDF schema relations.

The concise and human readable RDB-to-RDF/OWL mappings in a situation of an involved schema correspondence is essential e.g. for relational database semantic reengineering task, where a possibly legacy database is to be mapped to RDF/OWL. As an existing approach in this area Semantic SQL [41], still its relations to the open SPARQL standard, as well as its abilities to handle complex dependencies within a mapping are unclear.

Defining human readable mappings has been for long an issue within MOF-centered [42] model transformation community. A model transformation languages such as MOF QVT [43], MOLA [44] or AGG [45] (there are many other languages available) may be used for structural presentation of a mapping information; however, these languages are not generally designed to benefit from the mapping specifics that arise in RDB-to-RDF/OWL setting, partially due to simple target model structure (RDF triples). We note an interesting practical experience report of this kind in [36].

Human readable RDB-to-RDF/OWL mapping language can be appropriate as documentation means to describe database table and column correspondences to conceptual classes and attributes. This approach has notable advantages over textual and graphical documentation means. Possible advantages: documentation is human readable and formal at the same time, precise, machine processable (e.g., for validation, reporting, etc) and traceable in both directions.

2.3 Working examples

In this section, we introduce three examples that we will use to explain in detail the existing related RDB-to-RDF/OWL mapping approaches and the ideas of this work.

2.3.1 Mini-university example

We will explain various approaches for bridging relational databases and OWL/RDFS ontologies on base of a simple example database reflecting a study registration system and the related OWL ontology. The example is taken from [38]. Below **Fig.2** and **Fig.3** a sample database schema is provided, as well as, the corresponding ontology (OWL class *Thing* is omitted for simplicity). Note, that we do not focus on the integrity constraints in the OWL ontology here.

Observe the table splitting (*COURSE*, *TEACHER*) and table merging (*Person* from *STUDENT* and *TEACHER*) in the ontology using the subclass relations; the *PersonID* OWL class that is based on non-primary key columns in each of *Student* and *Teacher*

tables; and the n:n relation *takes* that reflects a student-to-course association that in the RDB is implemented using the *Registration* table.

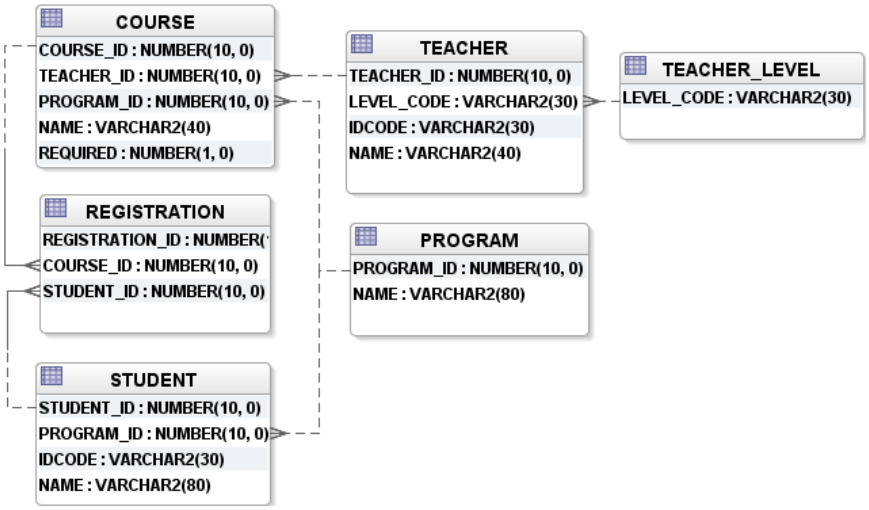


Figure 2. Mini-university relational database schema

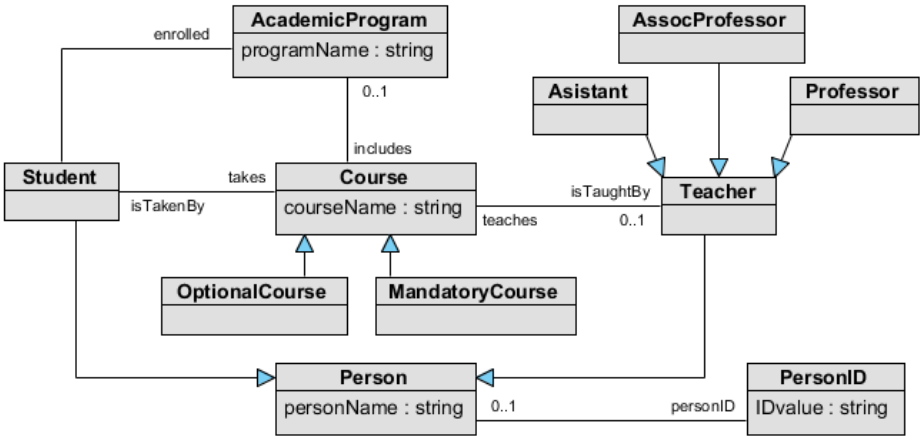


Figure 3. Mini-university ontology

For example, the *Student* and *Course* classes in this sample ontology correspond to the *student* and *course* tables in the sample database. To get instance data for *takes* OWL object property we need a table link path to join the *student* and *registration* tables on *student_id* column and *registration* and *course* tables- on *course_id* column.

Data for *personID* class instances comes from the *student* and *teacher* tables (*idcode* column). The *Asistant*, *AssocProfessor* and *Professor* classes all get instance data from the common *TEACHER* table but each has a different filtering expressed by

'level_code=...'. Similarly instances for two subclasses of *Course* class are determined by table row filtering *COURSE.required=1/0*.

In the following tables, we show the data in tables of our sample database. We will use this specific data set as an example.

Table 1. Table *program* data

program_id	name
1	Computer Science
2	Computer Engineering

Table 2. Table *teacher_level* data

level_code
Assistant
Associate Professor
Professor

Table 3. Table *course* data

course_id	name	program_id	teacher_id	Required
1	Programming Basics	2	3	0
2	Semantic Web	1	1	1
3	Computer Networks	2	2	1
4	Quantum Computations	1	2	0

Table 4. Table *student* data

student_id	name	idcode	program_id
1	Dave	123456789	1
2	Eve	987654321	2
3	Charlie	555555555	1
4	Ivan	345453432	2

Table 5. Table *teacher* data

teacher_id	name	idcode	level_code
1	Alice	999999999	Professor
2	Bob	777777777	Professor
3	Charlie	555555555	Assistant

Table 6. Table *registration* data

registration_id	student_id	course_id
1	1	2
2	1	4
3	2	1
4	2	3
5	3	2

2.3.2 Far table linking example

Suppose we have database with far linking tables with schema and data as below.

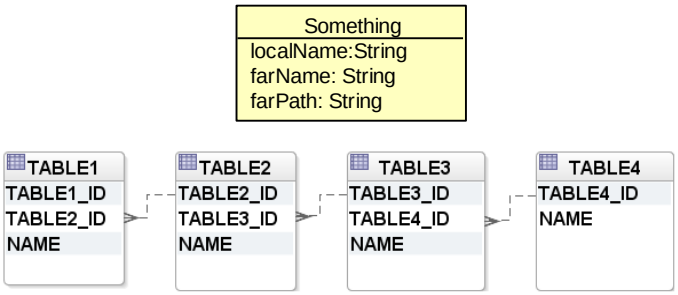


Figure 4. Far Table linking example- ontology and database schema

Table 7. Table *table1* data

table1_id	table2_id	name
1	1	table1 row1
2	2	table1 row2

Table 8. Table *table2* data

table2_id	table3_id	name
1	1	table2 row1
2	2	table2 row2

Table 9. Table *table3* data

table3_id	table4_id	name
1	1	table3 row1
2	2	table3 row2

Table 10. Table *table4* data

table4_id	name
1	table4 row1
2	table4 row2

We use table and column names without any semantics- only to illustrate the design pattern. The *table1* and *table2* tables are linked by two intermediate tables and four tables are involved in total. Suppose that OWL ontology consists of only one class *Something* that has three data properties: *localName, farName* and *farPath.* The class *Table* gets its data from database table *table1, localName* property gets data from *table1.name* field, property *farName* gets data from "far" table *table4* and property *farPath* gets data from *name* field of all tables on travel path from *table1* to *table4*.

2.3.3 Simple genealogy example

In this section, we illustrate D2RQ mapping for an example where two foreign keys based on the *father_id* and *mother_id* columns link a table to itself. **Figure 5** below shows this simple Database schema (one table only) and corresponding OWL ontology presented in MOF style.

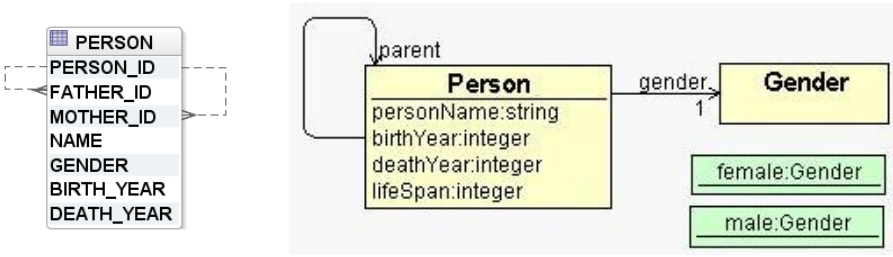


Figure 5. A RDB schema and ontology of simple Genealogy

Table below lists table data that corresponds to Adam and Eve's posterity taken from [46]

Table 11. Table *person* data (empty cells- null values)

person_id	father_id	mother_id	name	gender	birth-year	death_ year
1			Adam	m	0	930
2			Eve	f		
3	1	2	Cain	m		
4	1	2	Abel	m		
5	1	2	Seth	m	130	1042
6	5		Enos	m	235	1140
7	6		Cainan	m	325	1235
8	7		Mahalaleel	m	395	1290
9	7		Enan	m		
10	7		Mered	m		
11	7		Adah	m		
12	7		Zillah	m		
13	8		Jared	m	460	1422
14	11		Jabal	m		
15	11		Jubal	m		
16	12		Tubal-cain	m		
17	12		Naamah	m		

In this example, Database has higher granularity- for persons specified who the father or mother is. The ontology has only one object property *parent*. The information who mother or father is can be deduced:

```
Mother(x,y) ≡ parent(x,y) & gender(y)=female
Father(x,y) ≡ parent(x,y) & gender(y)=male
```

3 Existing RDB-to-RDF/OWL mapping approaches

In this section, we explain some of the existing RDB-to-RDF/OWL mapping approaches together with their benefits and shortcomings.

3.1 Direct mapping methods

3.1.1 A Direct Mapping of Relational Data to RDF (W3C)

There is candidate W3C standard for direct (technical) mapping [30] of RDBs to RDF format. Relational databases proliferate because they are efficient, have precise definition, have many SQL based tools and are most widespread comparing with other data technologies. The need has occurred to make data of relational databases globally accessible. One possible solution is to expose data in relational databases as RDF graphs that has web scalable architecture. The direct mapping defines a transformation from relational schema and data into RDF graph (called direct graph) whose target RDF vocabulary directly reflects the names of database schema elements.

The direct mapping takes into consideration the design of database tables and data they hold: tables, columns, primary/foreign key columns and data in row fields determine how to format IRI for triple parts (subject, predicate and object). For example, if table *X* has primary key consisting of n columns *C_1, C_2, ...,C_n* and row has values for these columns *V_1, V_2, ..., V_n* then subject IRI (called Row Node for a row) takes form:

```
base_IRI/X/C_1=V_1,C_2=V_2,...,C_n=V_n
```

We will not dive into technical details, they are described in document [**Error! Reference source not found.**]. Literal values of table row fields are transformed into triple as object value with predicate representing column name. Foreign key link is transformed into triple with subject and object containing Row DRF nodes of referenced rows on both sides. In addition, some corner cases of relational schema are considered e.g. foreign keys referencing candidate (unique) keys and hierarchical tables (sharing common primary keys). For tables that miss primary keys, blank nodes are created for Row RDF nodes.

We illustrate main points of direct mapping by fragment of direct graph for mini-university example [2.3.1] for the *Student* and *Program* tables:

```
@base <http://lumii.example/school/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


<Student/student_id=1> <rdf:type><Student> .
<Student/student_id=1> <Student#student_id>     1 .
<Student/student_id=1> <Student#name>     "Dave" .
<Student/student_id=1> <Student#idcode>"123456789" .
<Student/student_id=1> <Student#program_id>     <Program/program_id=1> .
```

```
<Student/student_id=2>  <Student#student_id>    2 .
<Student/student_id=2>  <Student#name>    "Eve" .
<Student/student_id=2>  <Student#idcode>  "987654321" .
<Student/student_id=2>  <Student#program_id>    <Program/program_id=2> .

<Student/student_id=3>  <Student#student_id>    3 .
<Student/student_id=3>  <Student#name>    "Charlie" .
<Student/student_id=3>  <Student#idcode>  "555555555" .
<Student/student_id=3>  <Student#program_id>    <Program/program_id=1> .

<Student/student_id=4>  <Student#student_id>    4 .
<Student/student_id=4>  <Student#name>    "Ivan" .
<Student/student_id=4>  <Student#idcode>  "345453432" .
<Student/student_id=4>  <Student#program_id>    <Program/program_id=2> .

<Program/program_id=1>  <rdf:type>  <Program> .
<Program/program_id=1>  < Program#program _id>  1 .
<Program/program_id=1>  < Program#name>   "Computer Science" .

<Program/program_id=2>  <rdf:type>  <Program> .
<Program/program_id=2>  < Program#program _id>  2 .
<Program/program_id=2>  < Program#name>   "Computer Engeneering" .
…
```

Notice that Relational.OWL platform uses similar approach as just described direct mapping. However, direct mapping as a W3C standard is desirable for integration purposes- many tools can be developed that transform relational data into RDF all complying with common W3C standard.

3.1.2 Relational.OWL platform

Cristian Perez de Laborda, Stefan Conrad from Heinrich-Heine Diseldorf University in DIGAME project (2004.) introduced a technology, which enables to represent relational schema/data as OWL ontology/RDF triples [20, 21]. Relational.OWL is a direct mapping platform that enables transformation of relational schema and data to OWL and RDF coding and therefore to be accessible from SPARQL endpoint.

The main purpose of the platform is to give means to look on relational data as RDF dataset and to query them by appropriate query language, e.g., SPARQL. Relational database data representation in RDF should correspond to relational model. For this purpose, a relational schema is described as OWL ontology. A central OWL ontology called *Relational.OWL* [49] serve as reference vocabulary. *Relational.OWL* ontology corresponds to relational schema metamodel. It contains classes *Database, Table, Column, PrimaryKey* and others; and it has OWL object properties such as *hasTable, hasColumn, isIdentifiedBy* and other. With the help of this central ontology a relational schema can be described.

Table 12. Correspondence between elements of Relational schema and Relational.OWL

element or relation of Relational schema	**element of Relational.OWL ontology**
Schema	Class *Database*
Table	Class *Table*
Table column	Class *Column*
Primary key	Class *PrimaryKey*
One element belongs to other	Property *has*
Table or Primary Key has column	Property *hasColumn*
Database contains table	Inverse functional property *hasTable* (a table can belong to no more than one relational schema)
Primary key attached to table	Functional property „isIdentifiedBy"
Foreign key	Functional property *references* (domain and range being class *Column*)
Maximal data length of column	Property *length*
Precision (decimal digits)	Property *scale*

The most often-used classes and properties of *Relationa.OWL* ontology are shown in picture below (taken from [50]):

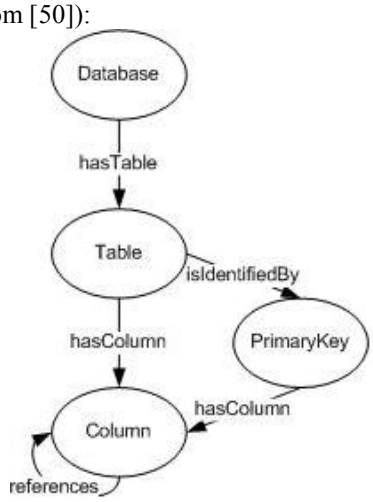


Figure 6. Relational.OWL Ontology

We note that *Relationa.OWL* ontology does not allow description of foreign keys based on more than one column because one column references one column. The *ForeignKey* class could be implemented similarly as the *PrimaryKey* class to allow of multiple-column foreign keys. Any relational schema can be expressed in OWL format by first describing namespace to Relationa.OWL and then the tables belonging to the database and the columns belonging to tables. See Appendix A.1.2 for

application of this approach to mini-university example and Appendix A.1.3 about RDB data transformation into RDF format thereof.

Benefits of Relational.OWL platform:

1. automatic transformation (ETL) of relational DB schema into OWL ontology denoted by ROWL (a technical one 1:1 corresponding to DB design);
2. automatic transformation (ETL) of relational data into RDF triple set that are instances of ontology ROWL;
3. capability to use SPARQL to query information about relational schema;
4. capability to use SPARQL to query relational data;
5. it provides a base for mapping relational schema(s) to target ontology by mapping ROWL to target ontology by SPARQL (this is described in Section 3.2.2).

3.1.3 DB2OWL- a tool for Automatic Ontology-to-Database Mapping

Nadine Cullot, Raji Ghawi, and Kokou Yétongnon from Universit de Bourgogne, Dijon, FRANCE in 2007 at Italian Symposium on Advanced Database Systems (SEBD 2007) presented RD2OWL tool for automatic Database-to-Ontology Mapping [59]. We will briefly describe the main points from this paper.

DB2OWL starting point is Relational Database schema structure- what are the tables, columns and relations between tables. A defined algorithm analyzes the structure of table model to infer an appropriate OWL ontology that conforms to the source database design.

Fig. 7 below (in taken from [60]) shows the architecture of DB2OWL implementation.

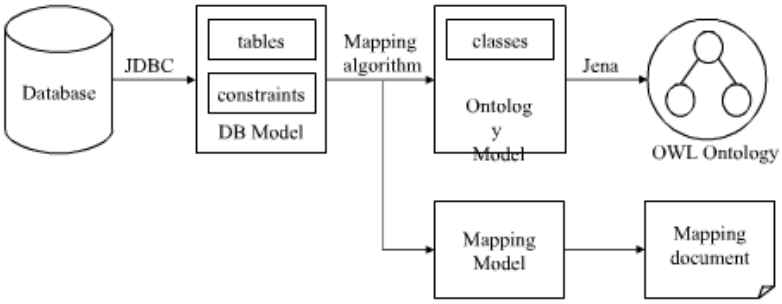


Fig. 7. RDB2OWL framework architecture

Mapping algorithm analyzes database tables and relation types between them and decides about corresponding OWL class and property creation. Three cases are considered:

1) Table T relate two other tables T1 and T2 in many-to-many relation;
2) Table T relate other table T1 by foreign key which is also primary key of T;
3) All other cases (not occurring case 1. or 2.).

To illustrate these cases we use database example from mini-university example (Section 2.3.1)

Table REGISTRATION is in case 1- it relates tables STUDENT and COURSE in many-to-many relation.

There are no tables in case 2. If there would be table PERSON with primary key column IDCODE and table STUDENT having IDCODE as primary key and also a foreign key to PERSON table then STUDENT table would be in case 2.

The following steps describe the mapping algorithm.

1. Database tables in case 3 are mapped to OWL classes.
2. The tables in case 2 are mapped to subclasses of classes corresponding to their related tables. For example, *STUDENT* table are mapped to subclass of a class mapped to *PERSON* table in case described above.
3. Tables in case 1 are mapped not to OWL classes but to the two object properties with domain and range determined by *T1* and *T2*. For example, for table *REGISTRATION* two object properties are created *student2course* and inverse *course2student* (*Student* and *Course* classes for domain and range).
4. If table *T* is in case 3 and relates to table *T1* by foreign key and c, c1 being classes corresponding to *T* and *T1* respectively. Create object property *op* that has domain c and range c1 and create inverse object property op' To preserve the original direction (from foreign key to primary) property op is marked as functional property. For example, for *TEACHER_ID* in *COURSE* table object properties *course2teacher* and *teacher2course* are created the first one being marked as functional.
5. For tables in case 2 that have other foreign key than the ones used to create the subclass, such key is mapped to object properties as in the previous step 4)
6. For all tables their columns that are not part of foreign keys are mapped to data properties.

Execution of the above-mentioned algorithm automatically generates a R2O [22] document that hold the generated mappings between original database and generated ontology. It can help to translate queries against generated ontology into SQL queries to retrieve corresponding instances.

The described in this section DB2OWL approach is not appropriate if source database is not well designed, (foreign/primary keys not explicitly defined, large tables corresponding to many concept, etc). We note that the correspondence of ontology generated with DB2OWL method is not so strictly conforming to relational model as the one generated with Relational.OWL approach [7].

3.1.4 Ultrawrap

Ultrawrap (Ultrawrap: Using SQL Views for RDB2RDF by Sequeda, J.F., Cunningham, C., Depena, R., Miranker, D.P. [26]) is a direct RDB-to-RDF mapping framework that enable dynamic SPARQL endpoint over data in legacy relational databases. The main benefits of Ultrawrap are:

1. Automatic publication of relational databases to Semantic Web;
2. Virtual RDF presentation by SPARQL-to-SQL transforming and execution on the fly;
3. Maximal use of existing SQL infrastructure and power of RDBMS.

The central element af Ultrawrap architecture is technical ontology called *Putative Ontology (PO)* that is obtained by syntactic translation of relational schema to OWL ontology (Tables correspond to ontology classes, table columns- to properties, etc). The RDF triples that correspond to PO ontology are implementated virtually as manually written three column SQL view *TripleView* over relational data for subject, predicate and object values. For our Mini-University database example, *TripleView* fragment may be written as follows (no target ontology names used, eg, „name" not „personName" property name used):

```
CREATE VIEW TripleView(s,p,o) AS
  SELECT 'Teacher' + t.teacher_id as s,
  'rdf:type'  as p, 'Teacher' as o
  FROM TEACHER t
UNION
SELECT 'Teacher' + teacher_id as s,
  'name' as p, t.name as o
  FROM TEACHER t
UNION
SELECT 'Student' + s.student_id as s,
  'rdf:type'  as p, 'Student' as o
  FROM STUDENT t
UNION
SELECT 'Student' + s.student_id as s,
  'name' as p, t.name as o
  FROM TEACHER t
UNION
...
```

Further, information from RDF triples is demanded as SPARQL query over *TripleView* which after syntax driven SPARQL-to-SQL translation is executed on relational database to get the live result. Example of SPARQL and corresponding SQL:

```
SPARQL:
SELECT ?person ?personName
WHERE { ?person rdf:type ?TEACHER.
        ?person name ?personName.
      }
SQL:
SELECT t1.s as person, t2.o as personName,
FROM tripleview t1, t2, t3
WHERE t1.p ='rdf:type'
AND t1.s = t2.s AND t2.p='name'
```

One of Ultrawrap priorities is performance therefore SQL optimizer techniques are used such as parametrized SQL queries and query rewrite by splitting queries into simpler ones. But SQL optimization techniques are somehow dependable on concrete RDBMS therefore ultrawrap may not be tuned well for all legacy databases. Users of Ultrawrap should be familiar with legacy database design in orded to manually formulate correct SPARQL queries.

3.2 Methods based on mappings

3.2.1 R2RML: RDB to RDF Mapping Language (W3C)

There is upcoming W3C standard R2RML [31] for RDB-to-RDF mapping language. The latest version at the time of writing is W3C Working Draft 24 March 2011. The purpose of R2RML language is to express customized mappings from relational databases to RDF datasets whose structure and target vocabulary can be chosen any, need not conform to relational schema. Vendors are welcomed to produce tools for R2RML language to enable view of relational data in RDF form for conceptual ontology.

R2RML mapping constructs specify how to produce RDF triple components subject, predicate and object in terms of source relational database table structure and data expressed in language structures. By table is meant a logical table- it can be view or SQL query. The main mapping constructs are:

- TriplesMap
  - SubjectMap
  - PredicateObjectMap
    - PredicateMap
    - ObjectMap
  - RefPredicateObjectMap
    - RefPredicateMap
    - RefObjectMap

It is clear from names of the constructs what they stand for. With *RefPredicateObjectMap* one can specify mapping for predicate, object pair that corresponds in database to link between two tables. We will illustrate the main features and design patterns of R2RML language with mappings fragments for mini-university example (see Section 2.3.1). For full mapping source code, see Appendix A.5.

If target ontology classes and properties correspond to database table and column, mappings are rather straightforward, e.g., for class *Program* and *programName* property:

```
<#TriplesMap_Program>
    a rr:TriplesMapClass;
    rr:tableName  "PROGRAM";

    rr:subjectMap [ rr:template "ex:program{program_id}";
                    rr:class ex:Program;
                  ];

    rr:predicateObjectMap
    [
      rr:predicateMap [ rr:predicate ex:programName ];
      rr:objectMap    [ rr:column "name"  ]
    ];
```

If correspondence is not expressible in terms of simple tables and columns references but involves row filters or calculated values then manual SQL coding is the

only solution. For example, *required* column of the *COURSE* table determine instance of which the *Course* subclass should be created (*OptionalCourse* or *MandatoryCourse*). The example shows dynamic instance type calculation (using *rdf:type* property):

```
<#TriplesMap_Course>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  course_id
            , teacher_id
            , program_id
            , name
            , case when required=1 then 'MandatoryCourse'
                   else 'OptionalCourse'
              end as subclass_name
      from    COURSE
    """;

    rr:subjectMap [ rr:template "ex:course{course_id}";
                    rr:class ex:Course;
                  ];

    rr:predicateObjectMap
    [
      rr:predicateMap [ rr:predicate rdf:type ];
      rr:objectMap    [ rr:template "ex:{subclass_name}"   ]
    ];
```

The language contains means to avoid duplicate coding. For example, the tripleMaps for all three subclasses of the *Teacher* class has the same predicateObjectMap. It can be defined in one place and reused several times:

```
<#PredicateObjectMap_personName>
    a rr:PredicateObjectMapClass
    [
      rr:predicateMap [ rr:predicate ex:personName ];
      rr:objectMap    [ rr:column "name"   ]
    ];
    .

<#TriplesMap_Assistant>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  teacher_id
            , name
      from    TEACHER
      where   level_code='Assistant'
    """;

    rr:subjectMap [ rr:template "ex:teacher{teacher_id}";
                    rr:class ex:Assistant;
                  ];

    rr:predicateObjectMap <#PredicateObjectMap_personName> ;
    .

<#TriplesMap_Professor>
…
<#TriplesMap_AssocProfessor>
…
```

R2RML has no means to express mapping for property that is based on link chain of more than two tables. Manual SQL coding can help in these cases (as for other non-direct mapping cases). For example for OWL object property *takes* (many to many relation between Student and Course classes) correspond link of three tables STUDENT-REGISTRATION- COURSE. Two TripleMaps are joined by means of course_id column from REGISTRATION table:

```
<#TriplesMap_Student>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  s.student_id
            , s.program_id
            , s.name
            , r.course_id
      from    STUDENT s, REGISTRATION r
      where   s.student_id=r.student_id
    """;

    rr:subjectMap [ rr:template "ex:student{student_id}";
                    rr:class ex:Student;
                  ];

    rr:refPredicateObjectMap
    [
        rr:refPredicateMap [  rr:predicate ex:takes ];
        rr:refObjectMap
        [
            rr:parentTriplesMap <#TriplesMap_Course>;
            rr:joinCondition
                "{childAlias.}course_id = {parentAlias.}course_id"
        ]
    ]
```

We note that R2RML is a rather low-level language for RDB-to-RDF mappings. Its expressiveness relies much on SQL language. In typical cases when data properties directly correspond to database table columns and object properties- to foreign keys, then it is possible to specify mappings without manual SQL coding. R2RML mapping language is not aware of target ontology therefore ontology structure (e.g., domain/range of properties) cannot be used to simplify mapping code or enhance expressiveness. We regard R2RML as a low level SQL oriented technical language. If R2RML eventually has supporting tools for mapping processing and triple generation then user oriented higher-order mapping languages may not need to have specific tools for triple generation. All they need is to have compiler for translation to R2RML.

3.2.2 Database to target OWL ontology mapping using Relational.OWL and SPARQL

We briefly describe in this section a technology to establish a mapping between relational Database and target ontology using RDF query language (SPARQL) presented by Christian Perez de Laborda and Stefan Conrad in their paper [20]. We will describe how to define these mapping. Examples in this Section and **Figure 8** are taken from [20].

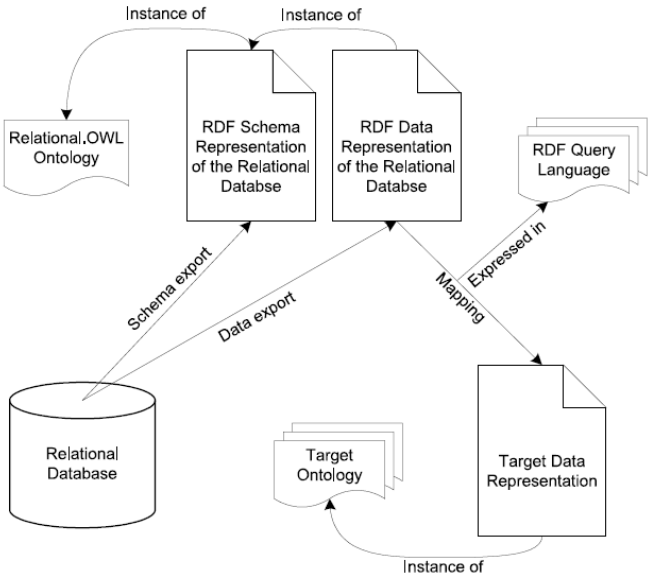


Figure 8. Mapping process

First Relational Database (RDB) schema is automatically exported into Relational.OWL representation denoted by ROWL (OWL ontology/RDF schema that is instance of Relational.OWL). Then relational data is automatically exported into RDF triple set of ROWL instances. This automatical export can be performed once by using Relational.OWL application [47] or on the fly by using RDQuery application [48]. This exported ontology lacks real semantics because it is in 1:1 correspondence with relational schema. Nevertheless, it can be queried by RDF query language (SPARQL) and analyzed by semantic Web reasoning tools, eg, Pellet [7]. To bring the relational schema/data into target ontology, mappings between the two ontologies need to be estasblished. This can be done by any RDF query language that is closed (meaning the resulting query response in valid RDF graph) SPARQL being appropriate for this. The mapping queries are to read ROWL ontology data and have to return data that is instance of target ontology. For details about these queries and for full query list for mini-university example see Appendix A.1.6.

RDQuery java application [48] enables some kind of online DB data retrieval. Relational data are not loaded into RDF store but SPARQL queries to read data from ROWL instances are translates into SQL online and executed:

- translate SPARQL query into SQL
- execute SQL in source database (only MySQL and IBM DB2 supported)
- translate SQL execution result back to RDF.

The approach of defining mapping as SPARQL queries has some drawbacks.

- First, writing SPARQL mapping queries can be a tedious manual effort.
- Second, SPARQL at present time is not as mature as SQL counterpart. It was not possible to use aggregations and function calls to express non-direct mappings. However, SPARQL develops, averages and function

calls are included into SPARQL v1.1 [51] and several tools support them in various degrees [52].

- If new blank nodes are created in one mapping script it is not possible to reference them from some other mapping script.

After translation relation schema and data to ROWL ontology, we have two ontologies- a technical one corresponding 1:1 to a database schema and a target one. There are various studies about mapping between ontologies. For example, Konstantinos Makris, Nikos Bikakis, Nektarios Gioldasis, Chrisa Tsinaraki, Stavros Christodoulakis in paper "Towards a Mediator based on OWL and SPARQL" [53] present a framework where mappings between ontologies are established and SPARQL queries reformulated.

3.2.3 R2O Database-to-ontology Mapping language and platform

R2O [Barrasa et al., 2006] [22] platform consists of declarative RDB-to-OWL mapping language and tools (Mapster) to process these mappings. RO2 language requires rather lengthy writing, but some help for code generation is in user interface of Mapster. R2O platform may be suitable for automatic generation of mapping code.

The mapping language is XML based and its syntax in BNF notion is available. The mappings describe how to obtain ontology class and property instances in terms of source database schema elements. ODEMapster [57] uses R2O mapping document to enable RDF triple generation in two possible modes: on-the-fly executing query or as a batch process to dump all triples in needed. The engine in implemented as a plug-in to NeOn toolkit [58] application, an Eclipse based Java desktop application. ODEMapster integration into Web applications enables them to provide global and semantic access to relational data through R2O mappings.

Both- a source database and target ontology are supposed to be independent (pre-existent). **Figure 9** below (taken from [22]) shows R2O mapping architecture.

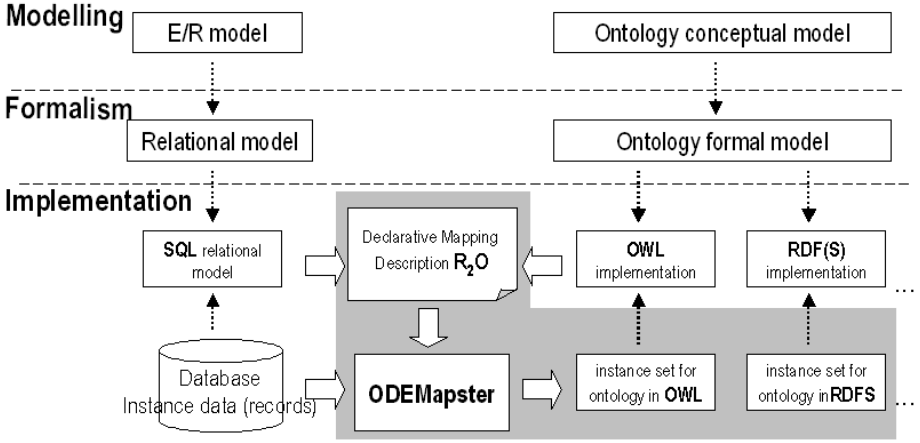


Figure 9. R2O mapping architecture

In R2O mapping the description of source RDB schema (can be more than one) is behind the <*dbschema-descr*> tag where information about relational DB tables, columns, primary/foreign key is described. Class-to-table mappings are described in element of <*conceptmap-def*> tag (corresponds to class map in D2RQ) with *name* parameter holding class name (full URI), <*uri-as*> describing how to generate instance URIs in terms of table columns, <*applies-if*> describing filter condition on row selection. Expressions are two type: transformations (used for URI formation and data property values) and condition expressions (e.g. for <*applies-if*>). All expressions are described in R2O language, having predefined list of conditions (*lo_than*, *lo_than_str*, *equals*, *equals_str*, *date_before*, *between*, etc) and predefined list of functions (*get_nth_char*, *get_substring*, *concat*, *Multiply_type*, etc) and list of logical operators (*AND*, *OR*). No SQL functions or expression is possible to invoke but combination and nesting of R2O built-in functions and operators allows users to define complex expressions for the mapping needs (conditions, value expressions, etc).

Mapping descriptions for OWL/RDFS properties are included in class mapping description of the respective domain class- as elements under the <*conceptmap-def*> sub-tags. Elements of the <*attributemap-def*> tags describe data properties and the <*dbrelationmap-def*> element- object properties. The <*joins-via*> tags introduce explicit table links, but these can be omitted if the mapped tables for the domain and range classes are linked by unique foreign key/primary key link in the database. This implicit link definition is possible because R2O language is aware of source database schema.

Figure 10 below shows a screenshot of ODEMapster window where UI defines part of mappings for mini-university example. Only simplest mapping could be defined this way, no UI support for filters (*applies-if*) and table links of more than two tables. Manual coding can deal with these and other custom mappings. ODEMapster supports a subset of the R2O language however, it did not open and execute custom mappings written manually.

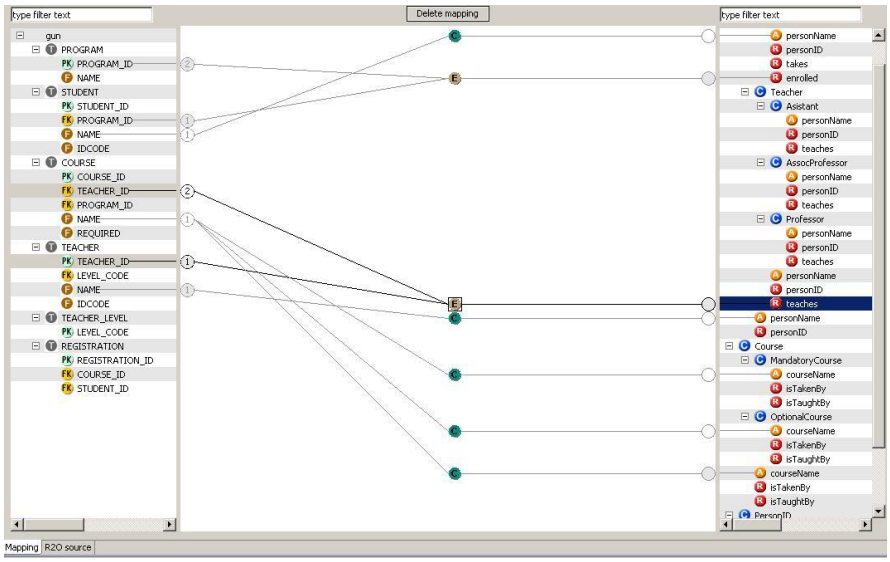


Figure 10. ODEMapster screenshot for simple university example mappings

We illustrate the main points of R2O mapping language by code fragments for mini-university example [2.3.1]. D2O mappings code is longer than respective D2RQ code for the same example. Full D2O code for the example is given in appendix A.4.

Mapping code begins with RDB schema description- tables, primary key columns (*keycol-desc*), foreign key columns (*forkeycol-desc*), regular columns (*nonkeycol-desc*)

```
<r2o>
  <dbschema-desc name="db">
    <has-table name="PROGRAM">
      <keycol-desc name="PROGRAM_ID"/>
      <nonkeycol-desc name="NAME"/>
    </has-table>
    <has-table name="STUDENT">
      <keycol-desc name="STUDENT_ID"/>
      <forkeycol-desc name="PROGRAM_ID">
        <refers-to>PROGRAM.PROGRAM_ID</refers-to>
      </forkeycol-desc>
      <nonkeycol-desc name="NAME"/>
      <nonkeycol-desc name="IDCODE"/>
    </has-table>
...
```

For each class of the ontology there is separate <*conceptmap-def*> section in the mapping code. Some has condition description <*applies-if*> (e.g., for classes *MandatoryCourse, OptionalCourse, Professor*). For the *Teacher* class the mapping (for instance and property values) expressions are:

```
<conceptmap-def name="http://lumii.lv/ex#Teacher">
  <uri-as>
    <operation oper-id="concat">
      <arg-restriction on-param="string1">
        <has-value>http://lumii.lv/ex#Teacher</has-value>
      </arg-restriction>
```

```
    <arg-restriction on-param="string2">
     <has-column>db.TEACHER.TEACHER_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <described-by>

   <attributemap-def name="http://lumii.lv/ex#personName">
    <selector>
     <aftertransform>
      <operation oper-id="constant">
       <arg-restriction on-param="const-val">
        <has-column>gun.TEACHER.NAME</has-column>
       </arg-restriction>
      </operation>
     </aftertransform>
    </selector>
   </attributemap-def>

   <dbrelationmap-def name=http://lumii.lv/ex#teaches
                      toConcept="http://lumii.lv/ex#Course">
    <joins-via>
     <condition oper-id="equals">
      <arg-restriction on-param="value1">
       <has-column>db.TEACHER.TEACHER_ID</has-column>
      </arg-restriction>
      <arg-restriction on-param="value2">
       <has-column>db.COURSE.TEACHER_ID</has-column>
      </arg-restriction>
     </condition>
    </joins-via>
   </dbrelationmap-def>
  </described-by>
 </conceptmap-def>
```

URI pattern definition from the above code shows native D2O function usage ("Teacher" concatenated with value of the *TEACHER.TEACHER_ID* field.) with no option for user-defined functions based on SQL expression. The *attributemap-def* tag describes mapping for the *personName* OWL data property.

The element of *db_relationmap-def* tag contain mapping code for OWL object properties with *toConcept* attribute pointing to the range class of the property and DB table joining conditions in the *joins-via* elements. The above code shows example of such mappings for the *teaches* OWL object property.

Note that <*conceptmap-def*> tags contain sub-tags <*attributemap-def*> and <*dbrelationmap-def*> meaning that the mapping codes for data and object properties are embedded into the mapping code for the domain class of the properties.

Mapping for the *takes* OWL object property shows an example how to define mapping to table links of more than two tables (*STUDENT→REGISTRATION→COURSE*). The *STUDENT* table (mapped to domain class) link to *COURSE* table (mapped to range class) through intermediate *REGISTRATION* table by means of two-column comparison (*equals*):

```
 <conceptmap-def name="http://lumii.lv/ex#Student">
…
   <dbrelationmap-def name="http://lumii.lv/ex#takes"
                      toConcept="http://lumii.lv/ex#Course">
    <joins-via>
     <AND>
```

```
      <condition oper-id="equals">
       <arg-restriction on-param="value1">
        <has-column>db.STUDENT.STUDENT_ID</has-column>
       </arg-restriction>
       <arg-restriction on-param="value2">
        <has-column>db.REGISTRATION.STUDENT_ID</has-column>
       </arg-restriction>
      </condition>
      <condition oper-id="equals">
       <arg-restriction on-param="value1">
        <has-column>db.REGISTRATION.COURSE_ID</has-column>
       </arg-restriction>
       <arg-restriction on-param="value2">
        <has-column>db.COURSE.COURSE_ID</has-column>
       </arg-restriction>
      </condition>
     </AND>
    </joins-via>
   </dbrelationmap-def>
  </described-by>
 </conceptmap-def>
```

For *Person* and *PersinID* class there are two mapping definitions by *conceptmap-def* (at end part of the script) for each as the two tables *STUDENT* and *TEACHER* both are the corresponding data tables for them.

```
<conceptmap-def name="http://lumii.lv/ex#Person">
…
  <described-by>
   <dbrelationmap-def name="http://lumii.lv/ex#personID"
                      toConcept="http://lumii.lv/ex#PersonID">
    <joins-via>
     <!-- Problem to make join as both classes for domain and range
          uses the same table.
          R2O language does not has facilities to assign
          aliases to tables
     -->
     <condition oper-id="equals">
      <arg-restriction on-param="value1">
       <has-column>db.STUDENT.STUDENT_ID</has-column>
      </arg-restriction>
      <arg-restriction on-param="value2">
       <has-column>db.STUDENT.STUDENT_ID</has-column>
      </arg-restriction>
     </condition>
    </joins-via>
   </dbrelationmap-def>
  </described-by>
 </conceptmap-def>
```

We noticed a problem in joining a table to itself- defining mappings for *personID* object property between classes *Person* and *PersonID*. It is up to tool implementation to manage such cases otherwise unwanted SQL may be generated:

```
select … from STUDENT, STUDENT
where STUDENT.STUDENT_ID=STUDENT.STUDENT_ID
```

We note that D2O mapping language use only functions and condition expressions defined on D2O language level. It means no possibility to use SQL expressions in mapping code for row filtering and property value calculation.

D2R mapping language is aware of RDB schema structure (has explicit description) and benefits from it. However, it is not aware of the target ontology and therefore, no domain/range information, subclass relation and other ontology information can facilitate the mapping specification.

We experienced that ODEMapster v.2.2.7 of 02.07.2010 [57] was not mature enough for real life applications- impossible to define row filters, far table links, instability (often crashes with uncaught java Exceptions). We did not find information about further development of the D2O language. Language specification available was from the original paper [22] from 2004.

3.2.4 D2RQ platform

D2RQ [23], [54] is platform- Treating Non-RDF Relational Databases as Virtual RDF Graphs (Prof. Dr. Chris Bizer form Freie Universität Berlin and colleges). It has a mapping language between RDB and OWL ontology and tools (D2R server) that process these mappings to enable SPARQL execution using relational data as virtual RDF graphs. The D2RQ mapping language is RDF based (typically written in n3 format) and is easier to write and more readable than R2O, supports any SQL expressions usage (not the case with R2O) but is not aware of the source RDB schema.

The initial version 0.1 of the framework is from June 2004, the current at the time of writing is version 0.7 from August 2009. D2RQ is a declarative language to define mappings between relational database schemas and OWL ontologies or RDF Schemas. **Figure 11** below (taken from [23]) shows the overall architecture of D2RQ platform.

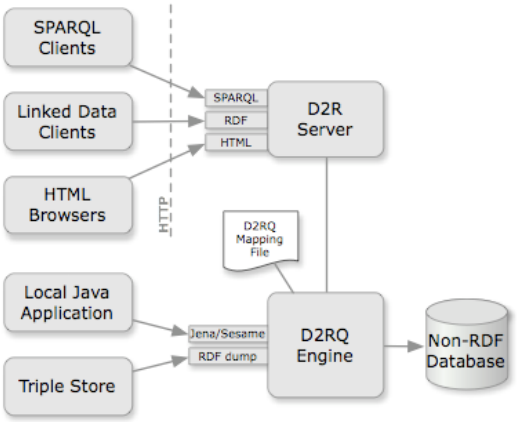


Figure 11. The architecture of the D2RQ Platform

The D2RQ platform has a declarative RDB-to-OWL maping language. Elementary D2RQ mapping expressions are expressed as RDF triples, and complete mapping document is RDF document typically in N3 serialization. D2RQ Engine is implemented as Jena [55] graph and enables using Jena or Sesame API [56] to get RDF triples by RDF graph processing or executing SPARQL queries. D2RQ

mappings enable translation of SPARQL queries to SQL and SQL execution result back as RDF graph. Web applications can provide SPARQL endpoint or RDF data browser by integration to D2R Server.

Benefits of D2RQ platform:

1. enables to declaratively define mapping between relational database schema and target ontology;
2. execution SPARQL queries over RDF- instance set of target ontology get data on-the-fly from relational database;
3. provides java API to use benefits 1 and 2 from java program code;
4. has web based SPARQL endpoint.

The code examples below show the main features of the D2RQ mapping language. We start with mini-university example.

D2RQ mapping scripts can specify one or more source relational databases for the target OWL/RDFS ontology, e.g.

```
map:database a d2rq:Database;
 d2rq:jdbcDriver "oracle.jdbc.driver.OracleDriver";
 d2rq:jdbcDSN "jdbc:oracle:thin:@guntars-PC:1521:gun";
 d2rq:username "school1";
 d2rq:password "s";
```

There are two type of map expressions: the ClassMap and PropertyBridge. The ClassMaps specifies how to get instance triples for OWL/RDFS classes. Mapping code below specifies correspondence of the *Course* class to all rows from *COURSE* table:

```
map:Course a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "course@@COURSE.COURSE_ID@@";
 d2rq:class ex:Course;
.
```

Pattern in *d2rq:uriPattern* specifies URI generation for subject and predicate part. There is no separate property in ClassMap where one can specify a table name for the class map. The table name in written in *d2rq:uriPattern* property value together with pattern expression (pattern between two "@@"). This makes the language more complicated.

The *PropertyBridge* specifies how to get instance triples for data or object properties. Property Bridge uses `d2rq:belongsToClassMap` to reference a defined classMap for domain class. Mapping code for OWL data property *className* specifies that value for the property be taken from the *COURSE.NAME* table column:

```
map:courseName a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:Course;
  d2rq:property ex:courseName;
  d2rq:column "COURSE.NAME";
  d2rq:datatype xsd:string;
  .
```

Object properties use also *d2rq:refersToClassMap* to reference a defined classMap for the range class. Mapping for OWL object property *teaches* specifies class maps for domain (belongsToClassMap) and range (refersToClassMap) based on tables *TEACHER* and COURSE respectively:

```
map:teaches a d2rq:PropertyBridge;
   d2rq:belongsToClassMap map:Teacher;
   d2rq:property ex:teaches;
   d2rq:refersToClassMap map:Course;
   d2rq:join "TEACHER.TEACHER_ID <= COURSE.TEACHER_ID";
```

.

Domain class maps specify the triple subject part generation for class instances, and range class maps- the object part generation for property values. Expressions in the `d2rq:join` or `d2rq.condition` elements specify conditions for table row links or filters.

The object property *takes* links classes *Student* and *Course* in many-to-many relation, while in the source database there are two linking steps *STUDENT→REGISTRATION→COURSE*:

```
map:takes a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:Student;
  d2rq:property ex:takes;
  d2rq:refersToClassMap map:Course;
  d2rq:join "XSTUDENT.STUDENT_ID <= XREGISTRATION.STUDENT_ID ";
  d2rq:join "XREGISTRATION.COURSE_ID => XCOURSE.COURSE_ID";
  .
```

One would ask why to define class maps for the *Course* superclass but not for the *MandatoryCourse* and *OptionalCourse* subclasses with appropriate filter expression for column *COURSE.REQUIRED*:

```
map:MandatoryCourse a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:uriPattern "course@@COURSE.COURSE_ID@@";
  d2rq:condition "required=1";
  d2rq:class ex:MandatoryCourse;
.
map:OptionalCourse a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:uriPattern "course@@COURSE.COURSE_ID@@";
  d2rq:condition "required=0";
  d2rq:class ex:OptionalCourse;
.
```

The answer is- to avoid duplicate class map and property map code blocks (*courseName* property for both subclasses *MandatoryCourse* and *OptionalCourse*). A better design pattern (taken from D2RQ documentation) states: specify class maps for sublcasses as property bridges for the *rdf:type* property referring to class map of common superclass *map:Course.* Row filtering expression determines the instances of each subclass. The same design pattern is appropriate for all three subclasses of the *Teacher* class (*Assistant*, *Professor* and *AssocProfessor*) as well.

```
# property bridge for OptionalCourse
map:OptionalCourse a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Course;
       d2rq:property rdf:type;
       d2rq:condition "required=0";
       d2rq:constantValue ex:OptionalCourse;
  .
# property bridge for MandatoryCourse class
map:MandatoryCourse a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Course;
       d2rq:property rdf:type;
       d2rq:condition "required=1";
       d2rq:constantValue ex:MandatoryCourse;
  .
```

One class may have more than one corresponding table (the *STUDENT* and *TEACHER* tables for the *PersonID* class) therefore, there are two separate class maps for each table and two property bridges for the *IDvalue* property:

```
# 1. class map for PersonID
map:PersonID_teacher a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:uriPattern "personID@@TEACHER.IDCODE@@";
  d2rq:class ex:PersonID;
  .
# 1. property bridge for IDValue
  map:IDValue1 a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:PersonID_teacher;
  d2rq:property ex:IDValue;
  d2rq:column "TEACHER.IDCODE";
  .
# 2. class map for PersonID
map:PersonID_student a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:uriPattern "personID@@STUDENT.IDCODE@@";
  d2rq:class ex:PersonID;
  .
# 2. property bridge for IDValue
map:IDValue2 a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:PersonID_student;
  d2rq:property ex:IDValue;
  d2rq:column "STUDENT.IDCODE";
```

Full D2RQ mapping code for mini-university example is given in Appendix A.2.

We will show D2RQ mappings for far table linking case, e.g., chain of more than two linked tables by 2 foreign key links correspond to OWL property. We use far-link database and target ontology example described in Section 2.3.2.

For chains of linked tables, each join is described by a separate *d2rq.joins* property that specifies linking conditions (column based). Value of OWL data properties may be mapped to a single table column (specified in *d2rq.column* parameter) or to a SQL expression (by *d2rq. sqlExpression* parameter).

The full mapping code for the far-link example is given in Appendix A.2.2. The code fragment below shows how to fill value of the *farName* data property with table field reachable by chain of several foreign key links starting from the current table. Property *farPath* records also all values from columns in all middle linking steps. The solution uses multiple *d2rq.join* parameters ("||" is SQL standard string concatenation operator):

```
map:ClassForTable a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "table@@TABLE1.TABLE1_ID@@";
 d2rq:class ex:ClassForTable;
 .
map:farName a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:ClassForTable;
    d2rq:property ex:farName;
    d2rq:join "TABLE1.TABLE2_ID => TABLE2.TABLE2_ID ";
    d2rq:join "TABLE2.TABLE3_ID => TABLE3.TABLE3_ID ";
    d2rq:join "TABLE3.TABLE4_ID => TABLE4.TABLE4_ID ";
    d2rq:column "TABLE4.NAME";
 .
map:farPath a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:ClassForTable;
    d2rq:property ex:farPath;
    d2rq:join "TABLE1.TABLE2_ID => TABLE2.TABLE2_ID ";
    d2rq:join "TABLE2.TABLE3_ID => TABLE3.TABLE3_ID ";
    d2rq:join "TABLE3.TABLE4_ID => TABLE4.TABLE4_ID ";
```

```
    d2rq:sqlExpression "TABLE1.NAME || '->' || TABLE2.NAME
 || '->' || TABLE3.NAME || '->' || TABLE4.NAME";
 .
```

The result of retrieving the triples by simple SPARQL in d2r Server:

```
PREFIX ex: http://lumii.lv/ex#
PREFIX db: http://localhost:2020/resource
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
SELECT ?s ?p ?o
WHERE
{
 ?s ?p ?o
}
```

Table 13. SPARQL execution result to get all triples for far table linking example

s	**p**	**o**
db:table1	rdf:type	ex:Table
db:table2	rdf:type	ex:Table
db:table1	ex:farPath	“table1 row1->table2 row1-> table3 row1->table4 row1”
db:table2	ex:farPath	“table1 row2->table2 row1->table3 row2 ->table4 row2”
db:table1	ex:farName	“table4 row1”
db:table2	ex:farName	“table4 row2”

The next code fragment shows D2RQ mappings for case when a table is linked to itself. The database and corresponding target ontology are from genealogy example described in Section 2.3.3. Two foreign keys by columns *father_id* and *mother_id* link the *PERSON* table to itself.

D2RQ mapping fragment for the example is shown below (full code is given in Appendix A.2.3). Property bridge for *lifespan* property shows SQL expression usage for value calculation.

```
# Person class
map:Person a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "person@@PERSON.PERSON_ID@@";
 d2rq:class ex:Person;
 .
# lifeSpan property
 map:lifeSpan a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:lifeSpan;
 d2rq:sqlExpression "PERSON.DEATH_YEAR - PERSON.BIRTH_YEAR";
 d2rq:datatype xsd:integer;
 .
```

There are two PropertyBridges for the *parent* object property, one for each of the two foreign keys *father_id* and *mother_id*. Those two PropertyBridges connects the same *Person* class (for domain and range) meaning that the same database table *person* is being joined to itself. This is done by using *d2rq.alias* to use different alias for the second reference to the *person* table. Then *d2rq.join* joins the *person* to itself. Such aliasing for joining is appropriate for tables mapped to range class (used in ClassMap referenced from *d2rq.refersToClassMap*). The table used in ClassMap for domain of the property (*d2rq.belongsToClassMap*) should not be changed by aliasing, because the table for domain ClassMap is starting point for linking.

```
map:parent_father a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
```

```
 d2rq:property ex:parent;
 d2rq:refersToClassMap map:Person;
 d2rq:alias "PERSON AS PARENT";
 d2rq:join "PERSON.FATHER_ID => PARENT.PERSON_ID ";
    .
map:parent_mother a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:parent;
 d2rq:refersToClassMap map:Person;
 d2rq:alias "PERSON AS PARENT";
 d2rq:join "PERSON.MOTHER_ID => PARENT.PERSON_ID ";
    .
```

The D2RQ mapping offer translation tables to specify translation of concrete values coming from database to other literal values. This feature is useful when the same semantic meaning has different literals in database and OWL/RDFS. A typical RDB-to-XSD translation for Boolean literals is 0→false^^xsd:boolean, 1→true^^xsd:boolean. In our example, character '*f*' denotes female gender in database but for OWL, we have *ex:female* object.

```
map:Gender a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriColumn "PERSON.GENDER";
 d2rq:containsDuplicates "true";
 d2rq:class ex:Gender;
 d2rq:translateWith map:GenderTable
 .
map:GenderTable a d2rq:TranslationTable;
d2rq:translation [ d2rq:databaseValue "f"; d2rq:rdfValue "ex:female";
];
d2rq:translation [ d2rq:databaseValue "m"; d2rq:rdfValue "ex:male"; ]
.
map:gender a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:gender;
 d2rq:refersToClassMap map:Gender;
.
```

We note that D2RQ has some drawbacks. D2RQ mapping is declarative language that specifies RDF triple generation in terms of relational database schema but is unaware of target ontology structure. Therefore, the mapping author should state explicitly that could otherwise be inferred from the target ontology. For example, for object property it is necessary to specify referenced ClassMaps for domain and range. If D2RQ mapping language would be aware of target ontology structure then clauses *d2rq.belongsToClassMap* and *d2rq.rfersToClassMap* would be odd in typical cases. The code fragment below shows these two potentially odd parameters, for the *enrolled* property has the *Student* and *AcademicProgram* as unique domain and range classes (each with unique ClassMap prescribed).

```
map:enrollod a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Student;
    d2rq:property ex:enrolled;
    d2rq:refersToClassMap map:AcademicProgram;
    d2rq:join "XSTUDENT.PROGRAM_ID => XPROGRAM.PROGRAM_ID ";
```

Another drawback of D2RQ platform is superfluous triple generation in the following scenarios. Suppose we have a class *C* with subclasses *C1*, *C2*, …,*Cn* and property *P* that have *C* as domain. There are two D2RQ mapping implementation options in this case.

The first way is to define several PropertyBridges for property *P*, one for each subclass *Ci,* by pointing to its ClassMap.

```
map:C1 a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "…";
 d2rq:class ex:Ci;
 .
map:P1 a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Ci;
       d2rq:property ex:P;
       d2rq:column "…";
.
```

This solution requires repeated typing: n times ClassMap and n times PropertyBridge.

Another mapping solution is to define one ClassMap for superclass *C,* define one PropertyBridge for property *p* and then define n PropertyBridges for *rdf:type* property instead of repeated references to ClassMaps of subclasses *C1, …,Cn* as done in the code fragment above.

```
map:C a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "…";
 d2rq:class ex:C;
 .
map:P a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:C;
       d2rq:property ex:P;
       d2rq:column "…";
.
```

For each i=1,2,…n

```
map:C1 a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Course;
       d2rq:property rdf:type;
       d2rq.condition "…"
       d2rq:constantValue ex:Ci;
```

This solution is more elegant than the first one (no repeated typing) but it generates superfluous triples for superclasses. It would be desirable if ClassMap had option to tell if the specified triples are to be physically generated. Then it would be possible to specify ClassMap for superclass *C* as not for triple generation (only for PropertyBridge reference).

Suppose, further that the target ontology has class *C* with many properties *P1*, *P2, …Pn*, having *C* as domain. Suppose also that ClassMap for the *C* class maps to source database table *T*, PropertyBridge for property *P1* maps to column *T.C1,* for property *P2* maps to column *T.C2*,… and for the property *Pn* maps to column *T.Cn*. In case when database schema design is not as we expect (legacy database), n can be large (>100). In this situation ClassMap for the *C,* class would normally generate instances for all rows of table *T* when no *d2rq.condition* is set. To specify condition when at least one property value triple exists for instance of *C*, one would specify lengthy condition. Without this tedious work, superfluous triples would be generated.

```
T.C1 is not null and T.C2 is not null … and T.Cn is not null
```

3.2.5 Virtuoso RDF views

Virtuoso RDF views (C. Blakeley, OpenLink Software, 2007, [24]) is framework that has a mapping specification language between relational database and target OWL/RDFS ontology and also has tools to process SPARQL queries accessing relational data on the fly through defined mappings. Virtuoso RDB-to-OWL mapping language is more complicated than R2O and D2RQ but provides additional option to specify more execution details (e.g. functions for URI patterns).

"Meta Schema Language" expresses relational database schema/data mapping to OWL/RDF. It is mapping definition language where quad map patterns are described by SPARQL notion. In typical cases, tables are mapped to RDFS classes, table columns to OWL data properties and foreign keys to OWL object properties. The mapping language is expressive enough to cope with non-direct custom mapping situations.

RDF datasets are not physically stored. Stored mapping definitions comprise so called RDF views over relational data. These mappings are calculated on-the-fly when triples are demanded. When relational data change, results of queries over target ontology through RDF views also change.

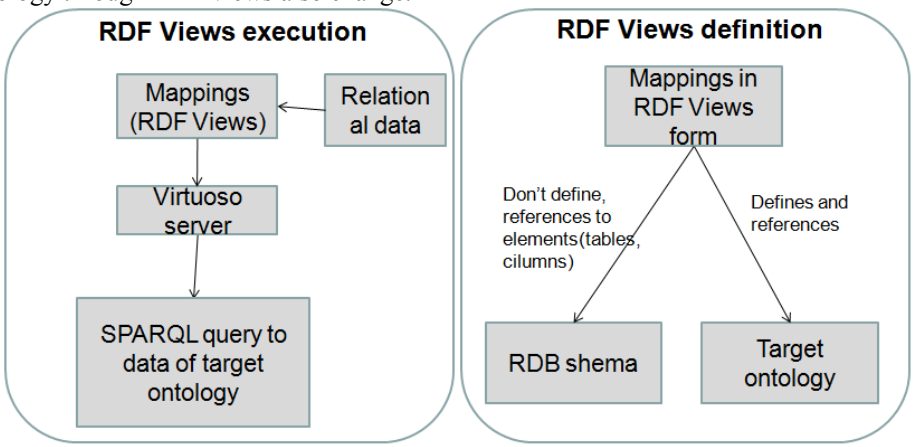


Figure 12. Virtuoso RDF views architecture

Below some features of the language are illustrated for mini-university example (Section 2.3.1). The full mapping code is given in Appendix A.3.

The mapping language has means to describe the target ontology. The code below shows description of RDF class, OWL data and object property as well as subclass relation:

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix virtrdf: <http://www.openlinksw.com/schemas/virtrdf#> .
@prefix DB: <http://lumiiex/school/> .

DB:Course a rdfs:Class .
```

```
DB:courseName a owl:DatatypeProperty .
DB:courseName rdfs:range xsd:string .
DB:courseName rdfs:domain DB:Course .

DB:isTaughtBy a owl:ObjectProperty .
DB:isTaughtBy rdfs:domain DB:Course .
DB:isTaughtBy rdfs:range DB:Teacher .

DB:MandatoryCourse a rdfs:Class .
DB:MandatoryCourse rdfs:subClassOf DB:Course .
…
```

IRI class constructs are responsible for subject URI generation for class instance triples (*rdf:type* predicate). The classes are like functions and uses c language format style for IRI patterns (%d- integer, %s- string, etc):

```
SPARQL
prefix DB: <http://lumiiex/school/>
create iri class DB:teacher_iri "http://lumiiex/school/teacher%d"
(in _TEACHER_ID numeric not null) . ;

SPARQL
prefix DB: <http://lumiiex/school/>
create iri class DB:course_iri "http://lumiiex/school/course%d"
(in _COURSE_ID numeric not null) . ;
…
```

The mapping specifications for classes and properties both are defined in one triple pattern) and reference the defined IRI patterns (course_iri). Code below specifies mappings for the *Course* class and *courseName* data property and *isTaughtBy* object property:

```
SPARQL
prefix DB: <http://lumiiex/school/>
create quad storage virtrdf:school
 from DB.DBA.COURSE as course_s
 from DB.DBA.TEACHER as teacher_s
   where (^{course_s.}^.TEACHER_ID = ^{teacher_s.}^.TEACHER_ID)
{
 create DB:qm-course as graph <http://lumiiex/school/#>
 {
  DB:course_iri(course_s.COURSE_ID)    a DB:Course ;
                DB:courseName    course_s.NAME ;
                DB:isTaughtBy    DB:teacher_iri(teacher_s.TEACHER_ID)
.
 }
};
```

Mapping for subclass (e.g. *MandatoryCourse* subclass of *Course*) typically reference the same URI pattern as the mapping for the superclass (*course_iri*). Additional SQL filter *course.required=1* filters out table rows for subclass:

```
SPARQL
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.COURSE as course_s_mand
   where (^{course_s_mand.}^.REQUIRED = 1)
{
 create DB:qm-mandatory_course as graph <http://lumiiex/school/#>
 {
  DB:course_iri (course_s_mand.COURSE_ID)  a DB:MandatoryCourse .
 }
};
```

Mapping for the OWL object property *takes* need to specify link chain of 3 tables *student→registration→course*. It is done in similarly as typically tables are joined in SQL code:

```
SPARQL
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.STUDENT as student_s
 from DB.DBA.REGISTRATION as registration_s
  where ( ^{registration_s.}^.STUDENT_ID = ^{student_s.}^.STUDENT_ID)
 from DB.DBA.COURSE as course_s_taken
  where ( ^{course_s_taken.}^.COURSE_ID =
^{registration_s.}^.COURSE_ID  )
{
 create DB:qm-student as graph  <http://lumiiex/school/#>
 {
  DB:student_iri(student_s.STUDENT_ID)  a DB:Student ;
     DB:personName student_s.NAME as DB:dba-student-name ;
     DB:takes DB:course_iri(registration_s.COURSE_ID) .
 }
};
```

To execute SPARQL queries using RDF views, SPARQL endpoint should be directed to Virtuoso server where mappings are loaded. The *input:storage* clause specifies over which named RDF Views SPARQL should be executed:

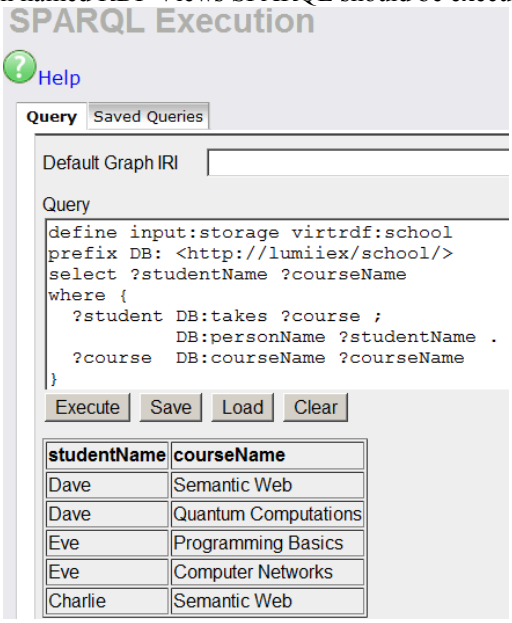


Figure 13. SPARQL execution over RDF Views

We make some notes about mapping language of Virtuoso RDF views. The mapping language is rather technical, it takes some effort to learn (We used only simplest constructions in the example codes above). Custom mapping are more complex when structures of source database schema and target ontology differs significantly. It is not clear how to use SQL expressions for value calculatuions for

data properties. For Virtuoso opensource version only built in database is available but commercial version allows connection to external RDBs that has jdbc driver.

3.2.6 Triplify

Triplify [27] is a simple approach to publish Linked Data from relational databases that are hidden behind web applications. Triplify offers an adapter (PHP) and configuration that can be integrated into existing web applications (added to web application root) to enable RDF and Linked Data generation by executing defined database SQL views.

The Triplify Web site [61] includes a repository of various Triplify configurations for popular Web applications part of which are contributed from third party companies. It shows wide usage of Triplify solution in the industry.

Triplify platform has no special mapping language but RDB-to-OWL mappings are expressed as SQL views with special structure (the first columns returns identifiers for instance URIs, columns names are for property URI generation, etc). Triples are demanded by means of HTTP requests with special URL patterns (for example, a Student instance URL could be in the form *http://lumii.ex/mini-university/triplify/student/3*). Triple extraction can be performed on demand or in ETL(Extract-Transform-Load) scenarios. For ETL performance improvement, special update logs are integrated to enable incremental RDF update.

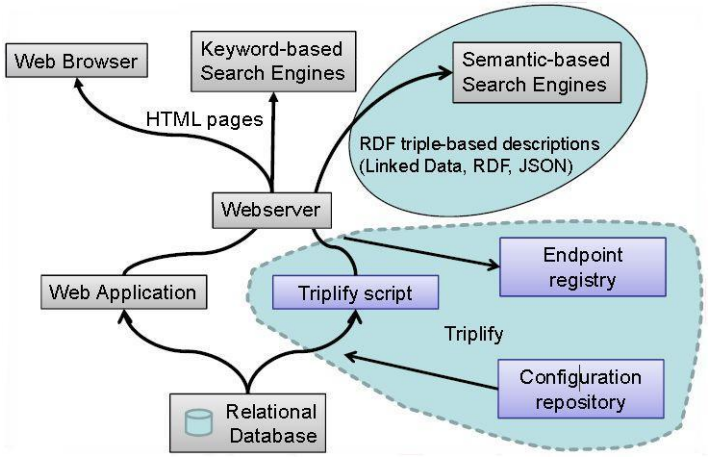


Figure 14. Triplify overview: the Triplify script with a configuration repository and an endpoint registry (picture taken from [27]).

The code below shows mapping from URL patterns to SQL query sets in the Triplify configuration (PHP code) for the Mini-University example fragment with instance RDF for the *Student* class and subclasses of *Course*, the *personName* and *courseName* data properties and *takes* object property. Mapping burden is laid on column naming (with arrow -> specifying object property instances creation):

```
$triplify['queries']=array(
  'student'=>array(
```

```
    "SELECT student_id,
      name AS 'personName^^xsd:string',
      'Student with name:' || name AS 'rdfs:label@en'
      FROM STUDENT ",
    "SELECT student_id,
      course_id AS 'takes->mandatoryCourse'
      FROM REGISTRATION r, course c ON c.course_id=r.course_id
      WHERE c.required=1",
    "SELECT student_id,
      course_id AS 'takes->optionalCourse'
      FROM REGISTRATION r, course c ON c.course_id=r.course_id
      WHERE c.required=0",
    ),
  'mandatoryCourse'=>
    "SELECT course_id,
      name AS 'personName^^xsd:string'
      FROM COURSE WHERE required=1",
  'optionalCourse'=>
    "SELECT course_id,
      name AS 'personName^^xsd:string'
      FROM COURSE WHERE required=0",
);
```

The Triplify mapping expressiveness and readability is the same as of SQL. SQL as a mapping language is appropriate for performance purpose: to generate more triples faster, benefitting from the power of modern relational databases. However, the week point is that SQL statements for triple generation have to be written manually.

3.2.7 DartGrid

Dartgrid is a Semantic Web toolkit [25, 40] for mapping and querying of relational database data as RDF using SPARQL language. Mappings are essentially table based where mapped elements from database and ontology are linked. There is no special mapping specification language. The mappings are defined by help of visual tool, which shows structures of source database schema and of target ontology and allows to link the corresponding items. Visual tools also help users to construct correct SPARQL queries. For query execution, SPARQL-to-SQL transformation algorithm uses mapping definitions. Datrtgrid tools offer an application development framework that allows interconnection of distributed relational databases for semantic querying, search and navigation services. The platform provides also full-text search capability with concept ranking.

3.2.8 Spyder tool

A new emerging mapping language approach is Spyder tool by Revelytix [62]. It is application that allows users to query the relational database in terms of target (domain) ontology with SPARQL. The mappings from source database to target ontology can be expressed in Revelytix RDB Mapping Language (native language) or in R2RML- the new W3C standard RDB-to-OWL mapping language. RDF triples can

be obtained from relational data on-the-fly therefore, changes in relational data are seen in subsequent SPARQL query executions.

Some of the Revelytix RDB Mapping Language [63] features are:

- describes the source relational database schema (this info is imported automatically);
- describes the target ontology;
- mapping expressions typically use explicit reference to both target ontology entities and source relational database constructs;
- can minimize repetitions (by using references to constructs, URI formats, etc);
- can use full power of SQL for complex transformations;
- simple mapping cases can be expressed simply;
- shorter forms of mapping expressions can be obtained by using implicit information (e.g., primary/foreign key column list)

The Revelytix RDB Mapping Language shares some ideas with D2RQ but differs from D2RQ in the details. As D2RQ, the Revelytix RDB mapping language specifications are expressed in RDF and written in N3 format. It has also MOF type metamodel that enable model usability. The structure of source database schema and target ontology are written in mapping specification document which allows to perform dependency and impact analyses relative to changes in the target ontology or source database schema. The mapping constructs allows using SQL expressions, which means that complex mapping cases, can be solved by SQL dialects provided by modern RDBMS. Example of Revelytix RDB Mapping Language for Mini-University example fragment (classes *Student*, *Course* and properties *courseName* and *takes* (the *domain:* prefix references target ontology entities and *db:* references elements of the source database schema) :

```
:StudentCM a map:ClassMap;
  map:source db:STUDENT;
  map:subjectString "ex:/student/STUDENT_ID";
  map:class domain:Student;

:CourseCM a map:ClassMap;
  map:source db:COURSE;
  map:subjectString "ex:/course/COURSE_ID";
  map:class domain:Course;

:courseNamePM a map:PropertyMap;
  map:propertyOf :CourseCM;
  map:predicate domain:courseName;
  map:literalValue db:name;

:takesPM a map:PropertyMap;
  map:propertyOf :StudentCM;
  map:source db:REGISTRATION;
  map:source db:COURSE;
  map:criteriaString " STUDENT.STUDENT_ID=REGISTRATION.STUDENT_ID
                       AND REGISTRATION.COURSE_ID=COURSE.COURSE_ID";
  map:predicate domain:takes;
  map:resourceValueString "ex:/course/<COURSE.COURSE_ID>".
```

We note that Revelytix RDB Mapping Language allows for shorter expressions by implicitly using information from database schema, for example, foreign key columns:

```
:enrolledPM a map:PropertyMap;
```

```
map:propertyOf :StudentCM;
map:predicate domain:enrolled;
map:subject db:student_program_FK.
```

Revelytix RDB Mapping Language references database schema objects and target ontology elements by links (URIs) allowing validation or analytics evaluations done by SPARQL (mapping specifications are RDF triples). We note that not all database references are expressed by links, for example, criteria strings or join expressions could be written as hand coded SQL text. Although Revelytix RDB mapping language is designed to be user friendly (shorter forms, fewer repetitions- fewer errors) it is rather complicated language because it tries to cover models as fully as possible (e.g., classes Table, Column, KeyColumn, PrimaryKey, and properties between them). The technically complicated part can be imported while mappings author writes references by hand.

3.2.9 Mapping Approach by Model Transformations

There is an approach based on using graphical model transformation language for RDB-to-RDF/OWL mapping solution in 2010 by S. Rikacovs, J. Barzdins [36]. In this approach RDF triples for target ontology are generated in multistep process:1) load source database schema and data into a meta-model repository; 2) load target ontology into the same meta-model repository as in previous step; 3) expert specifies mappings between elements of the source database schema and target ontology in graphical model transformation language MOLA [44]; 4) execution of the mappings generates RDF triples in meta-model repository; 5) export the generated triples into RDF database. The described process is RAM demanding because memory based meta-model repository is filled with data from the source database. The method was applied to migrate 3G of relational data from 6 Latvian medical registries [11, 12] to RDF triples. It took 1,5 hours and required 8Gb of RAM.

For the improved solution [37], the relational data are no longer loaded into meta-model repository but MOLA-to-SQL compilation generates SQL commands for triple generation in the source database on-the-fly. The improved method, as reported, was applied for one of 6 Latvian medical registries and showed 2 times better execution speed, and that its implementation was in progress.

We note that this approach of using graphical model transformation language MOLA actually offer high level constructs for mapping specification. However, a domain expert needs to learn MOLA language before he can specify database-to-ontology mappings. In addition, MOLA as general-purpose model transformation language does not utilize RDB-to-OWL specifics. The described approach shows us an example how general-purpose model transformation language can be used for RDF-to-RDF/OWL mapping solution.

3.3 Issues not considered in RDB-to-RDF/OWL mapping approaches

Existing RDB-to-RDF/OWL mapping approaches, for example, D2RQ, Virtuoso RDF Views are concentrating on efficient machine processing of the mappings,

preferably querying RDBs on-the-fly from an SPARQL-enabled endpoint. Much less attention, however, has been given to creating high-level mapping definitions that are oriented towards readability for a human being and that have a capacity to handle complex database-to-ontology/RDF schema relations.

Direct mapping approaches (Relational.OWL, DB2OWL) are good to automatically transform data from relational databases into RDF format but they are not sufficient when dealing with legacy databases (schema without explicit primary and foreign keys, several data items in one field, table/column names not representing conceptual model, etc)

RDB-to-RDF/OWL mapping languages of D2RQ, Virtuoso RDF Views and R2RML refer to database information in SQL strings without explicit links to the database schema elements. This makes almost impossible to do mapping code analysis or validation by machines with respect to database schema structure elements (e.g., which tables are not mapped to the target ontology classes, which non-key table columns are not mapped to data properties). Parsing of SQL strings are delegated to RDBMS therefore, it is not possible to find syntax errors before execution of the mappings to generate RDF triples.

The mapping languages of D2RQ, Virtuoso RDF Views and R2RML are not aware of the source database schema structure (as exceptions can be mentioned R2O and Revelytix RDB Mapping Language). Therefore mapping author needs to repeat in the mapping code the information that can be obtained from the database schema (e.g. primary key columns; how foreign keys join tables).

R2O, D2RQ and R2RML mapping languages do not use information about structure of the target ontology (e.g., subclass relation) and therefore mapping author needs to repeat that information. For example, information about domain class of some property could allow not specifying the predicate part for that property mapping (e.g. not specify *belongsToClassMap* in D2RQ *propertyBridge* specification)

In other RDB-to-RDF/OWL mapping languages, high-level language construct such as user-defined functions (scalar and aggregate) for class-to-table joining and value calculation is not used. This can lead to repetitions in mapping code.

Table 14. Presence of features in existing mapping approaches: awareness of source database and target ontology structures and availability of high-level language constructs

	Aware of source database schema	**Aware of target ontology structure**	**High level language constructs**	**Mapping method**
A Direct Mapping of Relational Data to RDF (W3C candidate recommendation)	**no**	**no**	**no**	specifies format of direct RDF triples in terms of database schema
Relational.OWL	**yes**	**no**	**no**	automatic transformation

DB2OWL	**yes**	**no**	**no**	algorithm generates ontology by analyzing the source RDB schema structure
Ultrawrap	**yes**	**no**	**no**	by syntax driven RDB schema-to-OWL translation
R2RML	**no**	**no**	**no**	mapping language
Database to target OWL ontology mapping using Relational.OWL and SPARQL	**yes**	**no**	**no**	mappings expressed by manually written SPARQL
D2RQ	**no**	**no**	**no**	mapping language
R2O	**yes**	**no** in mapping language, but **yes-** aware in visual tool (Mapster)	**no**	mapping language
Virtuoso RDF views	**no**	**yes**	**partly** IRI class definition is like function definition that can be referenced (invoked) from other places	mapping language
Triplify	**no**	**no**	**no**	mappings expressed by manually written SQLs
DartGrid	**yes** not by language but by visual tool	**yes** not by language but by visual tool	**no**	expressed essentially by tables which can be filled by visual tool

Spyder tool	**yes**	**partly yes** target ontology structure elements not described in mapping language itself but are referenced in mapping code	**no**	mapping language: native is Revelytix RDB Mapping Language but support is for R2RML language as well
Mapping Approach by Model Transformations	**partly** RDB schema is automatically loaded in repository	**partly** ontology is automatically loaded in repository	**no** high level is model transformation language used for mapping code	general purpose graphical model transformation language as mapping means

As can be seen from **Table 14** the issue of high level functions and awareness of source and target model structures in RDB-to-RDF/OWL mapping constructions has not been the primary focus of research and implementation, still, as practical examples show, these constructions can be very useful in concrete practical mapping definitions.

4 RDB2OWL mapping specification language

RDB2OWL is a high level declarative RDB-to-RDF/OWL mapping specification language that offers high level language constructs and is aware of source database schema and target ontology structures and. As described in Section 3.3 there is need for such a mapping language.

High-level goals of the RDB2OWL mapping language are:

- define how classes and properties of target ontology are related to metadata constructs in source relational database;
- define how RDF triples that correspond to target ontology are generated from data in source relational database;
- explicit reference usage to elements of source database schema and entities of target ontology;
- provide means for avoiding repetitions (e.g., by referencing named class maps, usage of implicit foreign key information, use of functions);
- support for some mapping patterns occurring in real life cases.

RDB2OWL mapping language features are:

- reuse of RDB table column and key information, whenever that is available,
- concrete human readable syntax for mapping expressions that is very simple and intuitive in the simple cases, and can also handle more advanced cases,
- built-in and user defined functions (including column-valued functions and aggregate functions),
- advanced mapping definition primitives, e.g. multiclass conceptualization that avoids the need of specifying long filtering conditions arising due to fixing a missing conceptual structure on large database tables,
- possibility to resort to auxiliary structures defined on SQL level (e.g. user defined permanent and temporary tables, as well as SQL views), still maintaining the principle that the source RDB is to be kept read only.

RDB2OLWL mapping language is designed with primary aim to be user readable and capable to deal with mapping patterns occurring in real life examples. It is a high-level and declarative RDB-to-RDF/OWL mapping specification language that is based on re-using the target ontology structure as a backbone where mapping expressions are written as annotations (we use *DBExpr* annotations) to OWL ontology classes and properties, as well as to ontology itself (in most places OWL can be substituted with RDFS). Typical usage of *DBExpr* annotations are as follows. When a mapping expression *expr* specifies that OWL class *C* is mapped to a database table we write *expr* as *DBExpr* annotation to class *C*. When a mapping expression *expr* specifies a correspondence of OWL data property *p* to database table column we write *expr* as an annotation to the property *p*. When *expr* specifies correspondence of OWL object property *p* to relation between tables (e.g. foreign key) we write *expr* as an annotation to the property *p*.

RDB2OWL mapping language has grammar and MOF-style mapping metamodel (that can be re-phrased easily also into a mapping OWL ontology). We design RDB2OWL implementation as a relational database schema (RDB2OWL mapping RDB schema) where mappings are stored and executed by means of automatically

generating SQL statements that create (dump) RDF triples corresponding to the target OWL ontology from the source RDB data.

RDB2OWL mapping language is designed to be compilable to RDB2OWL mapping RDB schema (described in Section 5.2) for execution by multistep process (parsing mapping expressions into instances of mapping metamodel, then applying transformation steps and finally transformation into mapping RDB schema). RDB2OWL mapping expressions that are declarative and high level could be compiled also to other mapping languages such as in D2RQ [23], Virtuoso RDF Views [24] or R2RML [31] in order to use tools that support them or will support in the future.

The simple structure of the RDB2OWL mapping metamodel allows for treating its models also as documentation of the correspondence between the RDB and RDF/OWL schemas (accessible at least to technically literate user). This is important when the semantically re-engineered RDF/OWL models are themselves regarded as user-level documentation of the technical RDB schemas.

We note that our approach is not looking for automated mapping generation from field-to-property correspondences in the style of CLIO [64] (on a practical note, we need a richer join filtering language than CLIO permits). We are not primarily looking at applying the defined mappings in retrieving the data from source RDB on-the-fly when the data are requested by queries in a RDF model environment, as in platforms of D2RQ [23] or Virtuosi Views [24]. This saves us at least the considerations for efficiency of integrating queries over RDB into those over RDF data stores, as well as, allows for a greater freedom in mapping construction techniques. The closest approach to ours is that of R2O [22], where the same principal schema of employing the SQL engine for implementing the declaratively specified mappings is used.

We identify a few typical mapping patterns and propose solutions for their transparent (user-friendly) encoding into a mapping definition, including the cases when this leads to "meta-level" operations over the RDB schema and/or OWL ontology definition (e.g., analyzing all properties with a fixed specified domain, necessary to succinctly reflect the conceptualization by means of subclasses; or meta-level information tables for grouping table fields into a single multi-valued data or object property). Yet another "non-common" point in RDB2OWL is "virtual" class-to-table mappings that do not generate class instances, but can be referenced from object or data property mappings.

We divide the RDB2OWL mapping language into 3 levels:

- The RDB2OWL Raw level contains the basic language constructs.
- The RDB2OWL Core include additional constructs that allows to write mapping expressions concise omitting information that can be deduced from model structures of the source database schema and target ontology (e.g., foreign key column names).
- The RDB2OWL Core Plus contains additional advanced constructs (e.g., function definition, introduction of auxiliary database objects).

4.1 RDB2OWL Raw Mapping Language

4.1.1 RDB2OWL Raw metamodel

A RDB2OWL mapping is a relation between a source relational database schema ***S*** and target OWL ontology ***O***. The mapping specifies the correspondence between the concrete source database data (values in table row cell) and RDF triples "conforming" to the target ontology. We present the abstract syntax structure of a raw RDB2OWL mapping in a form of MOF-style [42] metamodel in **Figure 15**, with additional expression and filter metamodel in **Figure 16**.

The metamodel refers to RDB schema and OWL ontology structure descriptions, presented here as the RDB MM fragment and the OWL metamodel fragment. The RDB2OWL mapping classes themselves are in the middle part of **Figure 15**.

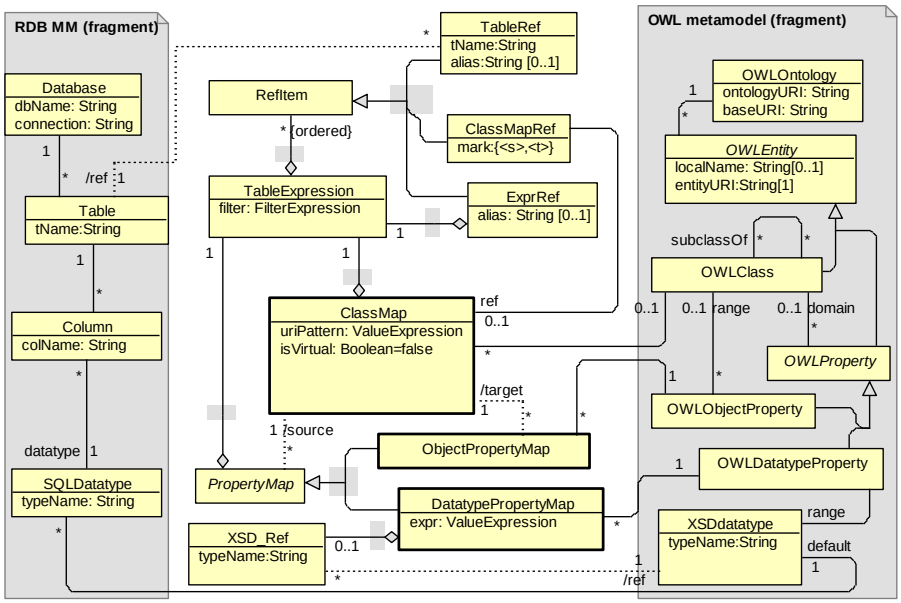


Figure 15. Raw RDB2OWL Raw mapping metamodel

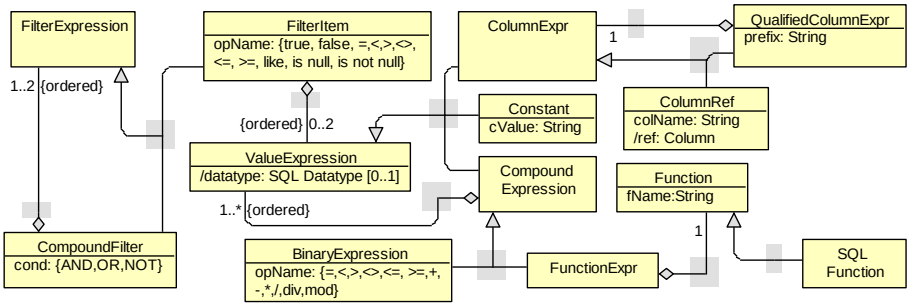


Figure 16. Expression and filter metamodel

An RDB2OWL mapping consists of "elementary mappings", or maps, that are instances of *ClassMap*, *ObjectPropertyMap* and *DatatypePropertyMap* classes in the mapping metamodel. In typical use cases the class maps (*ClassMap* instances) are responsible for Table-to-OWL Class mappings (with options to add filtering expressions and linked tables); data property maps (*DatatypePropertyMap* instances) provide Column-to-OWL DatatypeProperty mappings and object property maps (*ObjectPropertyMap* instances) establish OWL object property links that correspond to related tables in the database. In non-standard mapping patterns, for example, a class map can be defined without linked OWL class but is referenced from property maps; a data property maps may be based on any value expression and not merely on a single table column. We show the standard and non-standard mapping techniques in subsequent examples.

A class map (*ClassMap* instances) establish link from OWL Class to database table context using *TableExpression* instance in order to produce information necessary for RDF triples generation for OWL class instances . When OWL class *C* is linked to a database table *T* with row filtering expression *F*, then the following instances and links define the mapping:

`OWLClass(localName=C)→ ClassMap→`
`TableExpression(filter=F)→ TableRef→ Table(tName=T)`

An example of class map description

`COURSE c; c.required=1; {uri=('Student', student)}`

has a table expression `COURSE c; c.required=1` corresponding to the *TableExpression* class in metamodel and Uri pattern corresponding to the *ClassMap.uriPattern* attribute in the metamodel. For expression `COURSE c` there is *TableRef* class with attribute values *tName="COURSE"* and *alias="c"*.

If an OWL class is mapped to a table context consisting of several tables then class map's *TableExpression* instance contains more than one *RefItem* instances of *TableRef* type, each referencing one table of the context. Filtering expression in the *filter* attribute joins the tables. The table linking resemble how SQL statements introduce several tables with optional aliases in the FROM clauses and join them in the WHERE clauses.

The *ExprRef* class enables to build nested table expressions (*TableExpression→ ExprRef→ TableExpression*). Nesting table expressions are described in Section 4.1.2 about RDB2OWL syntax and they are used in transformation steps from RDB2OWL Core to RDB2OWL Raw (see Page 78).

The class maps that are denoted as virtual (*isVirtual=true*) are not used for the RDF triple generation themselves, still, they can be referred to from object and data property maps. In order to obtain RDF triples for an OWL class *C* we require that *OWLClass* instance for the *C* class (*localName*="*C*") should be linked to at least one non-virtual class map.

The data property maps (*DatatypePropertyMap* instances) provide Column-to-OWL DatatypeProperty mappings in typical cases and value expression-to-OWL DatatypeProperty in more general cases. Value of attribute *expr* is built according to expression and filter metamodel of **Figure 16**, and it specifies value calculation for OWL data property. Each data property map is based on a source class map (linked to the *source* link end) and can access the class map's table information; it can introduce further linked tables and filters into the table context for column expression evaluation by *TableExpression* instance attached directly to *DatatypePropertyMap*.

For example, data property description for *name* property with domain *Course:*

```
<s>.name^^xsd:String
```

the `<s>` denotes source class map (corresponds to *source* link end) and for `name` there is *expr* attribute of *DatatypePropertyMap* to be evaluated in source class map's table expression (COURSE table). For `xsd:String` there is *XSD_ref.typeName* class attribute in metamodel.

The object property maps (*ObjectPropertyMap* instances) establish OWL object property links that correspond to related tables in the database. The tables to be related generally come from the source and target class maps (*source* and *target* association ends) of the object property map; they are joined using explicit join condition specification in the object property map's table expression's *filter* attribute, with option to include further linked tables and filters through *TableExpression* instance attached directly to *ObjectPropertyMap*.

As an example, object property description for property *teaches* (domain class is *Teacher* and range class is *Course*) is:

```
<s>, <t>; <s>.teacher_id = <t>.teacher_id
```

The `<s>` and `<t>` stand for source and target class maps pointing to by *source* and *target* links in metamodel. Filter expression `<s>.teacher_id=<t>.teacher_id` corresponds to the *filter* attribute of the *TableExpression* class instance attached to the *PropertyMap* instance for the property.

For an OWL data or object property *p* a property map *m* (linked to *p*) has a *source* link (*PropertyMap*→*ClassMap*) to class map that is responsible for subject part generation of the RDF triples for property p. For an OWL object property *p* a property map *m* (linked to *p*) has also a *target* link (*ObjectPropertyMap*→*ClassMap*) to a class map that is responsible for object part generation of the RDF triples for property p. For any property map, we call these class maps as *source class map* and *target class map*.

In typical situations, these class maps can be deduced from ontology structure when the source class map resp. range class map is the only class map that is ascribed to property's domain class resp. range class. If this detection is not possible, (e.g., OWL property has no named class as its domain or range) then the mapping's author can explicitly define the source or target class maps and link them to the property map.

For a data property map *x* and its source class map *s* we require that the table expression attached to *x* has a class map reference (a *ClassMapRef* instance) with mark '<s>' that points to *s* as *ref*.

Similarly, we require for an object property map *x* and its source and target class maps *s* and *t* that the table expression attached to *x* has a class map reference (an *ClassMapRef* instance) with mark '<s>' that points to *s* as *ref*, and class map reference with mark '<t>' that points to *t* as *ref*.

We note that in the RDB2OWL Raw metamodel only the table and column structure of the source RDB is reflected, disregarding any primary and foreign key information (this information will be used in RDB2OWL Core language described in Section 4.2 in order to provide a more succinct mapping specification). Therefore, all information about table linking has to be stated explicitly in the mapping expressions. This approach is appropriate for legacy databases without presuming any normalization features in them.

The OWL metamodel fragment includes domain resp. range information for an OWL object or data property, if the property can be identified to have a single domain resp. range that is a named class or a data range (a subset of a known datatype). In the case of raw mapping, the only "structure" from the OWL part needed is URI associated to OWL entities (OWL classes, OWL data and object properties). For the advanced mapping features (see RDB2OWL Core Plus described in Section 4.3), we include, in addition, the (optional) *subclassOf* information for OWL classes, as well as domain information for OWL properties and range information for OWL object properties.

4.1.2 RDB2OWL Raw syntax

Syntactically the RDB2OWL mapping definition is achieved by storing textual class map and property map descriptions in the annotations to the respective OWL classes and properties (we assume a fixed annotation property *DBExpr* is used for this purpose). An OWL class may have several annotations each describing a class map; an OWL data or object property may have several annotations describing data or object property maps respectively. The syntax structure resembles RDB2OWL Raw metamodel of **Figure 15**. We present the grammar in simplified form for readability purpose. The detail grammar written in ALTLRWorks [74] tool is given in Appendix A.7.

We start mapping syntax explanation by table expressions that is the base for table context definition for class maps and property maps. A table expression description consists of a comma-separated list of reference items, followed by optional filter expression that is separated from the reference item list by a semicolon. Each reference item can be:

(i) a table name possibly followed by an alias,
(ii) a class map reference (one of strings '<*s*>' or '<*t*>', optionally preceded by a class map description), or
(iii) a table expression enclosed in parentheses and possibly followed by an alias string.

The filter expression is built in accordance to the abstract syntax of expression and filter metamodel of **Figure 16**. We created the syntax diagrams for EBNF expressions in ALTLRWorks tool [74] and use them further on to illustrate the RDB2OWL syntax.

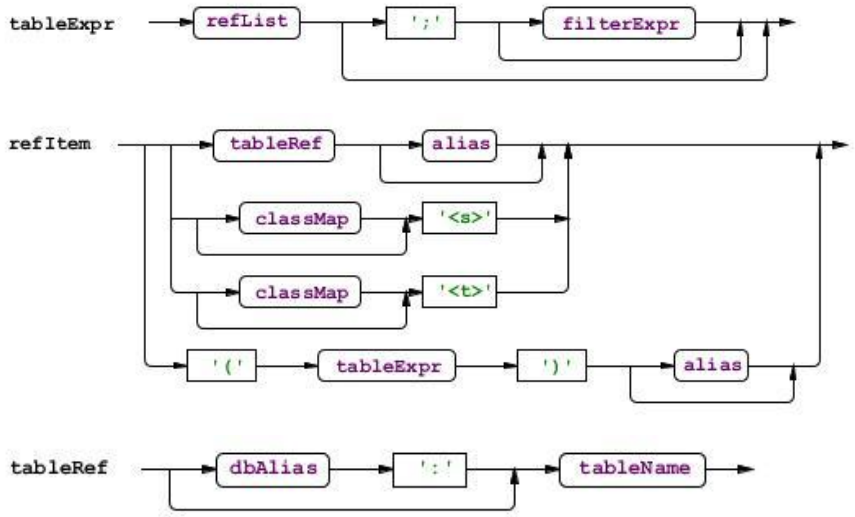


Figure 17. RDB2OWL Raw table expression syntax

The concrete syntax of expressions is based on SQL expression syntax, however not including SQL-style sub-queries. Since the RDB2OWL table expressions form a hierarchical structure where every hierarchy level can be identified by an alias

`tableExpr→refItem→(tableExpr) alias`

We let the *fully qualified names* (*fqn*, for short) for columns to be expressions of the form

$a_n.(a_{n-1}.(a_{n-2}. \ldots .(a_1.(a_0.c))\ldots)),$

where c is a column name of source database table, a_0 is a table name and $a_1 \ldots a_n$ are prefixes; each prefix is an alias or a class map reference mark. We let a column be identified within a table context not only by its *fqn*, but also by shorter forms (some of prefixes omitted) if that allows unique column identification.

Presenting the syntax, we presume the use of parentheses to allow unique identification of abstract syntax structure from the expression text. Some table expression examples are:

`STUDENT`	A simple table expression referencing one table without alias and filter
`STUDENT S, REGISTRATION R;` `S.student_id=R.student_id`	A table expression referencing 2 tables and joining them on equality of columns in filter expression

`STUDENT S, REGISTRATION R, COURSE C;`
`S.student_id=R.student_id`
`AND R.course_id=C.course_id`
	A table expression referencing 3 tables and joining them on equality of columns in filter expression

(STUDENT S, REGISTRATION R;
  S.student_id=R.student_id
) SR,
(TEACHER T, COURSE C;
  T.teacher_id=C.teacher_id
);
SR.(R.course_id)= COURSE.course_id
	Two nested table expressions are used inline in higher level table expression and using fully qualified names are used in filter of outer table expression to reference columns in the nested table expressions

<s>, <t>; <s>.teacher_id = <t>.teacher_id	This table expression contains a source class map references by <s> mark and target class map reference by <t>.

A class map description is obtained by adding to a table expression description an Uri pattern description in the form *{uri=(<item$_1$>,…,<item$_k$>)}*, where each *item$_i$* is a value expression (typically, a textual constant, or a reference to a database table column). Such pattern describes a conversion to Uri form and concatenation of all values *item$_i$*. If *'!No'* decoration is added at the end of class map description it means that the class map is **virtual**- no instance triples should be generated from it.

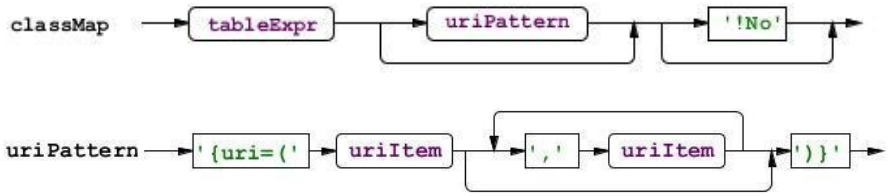


Figure 18. RDB2OWL Raw class map syntax

Some class map examples are (the first two are equivalent- the second used explicit SQL concatenation operator ||:

STUDENT {uri=('Person',student_id)}	A class map consisting of simple table expression referencing one table without alias and filters and having Uri pattern that states implicitly- concatenate the *'Person'* constant and value of column *student_id*.
STUDENT {uri=('Person' || student_id)}	The same as the above example with only difference that concatenation for Uri pattern is stated explicitly.
STUDENT S, PROGRAM P; S.program_id=P.program_id {uri=('program_' || P.pname, '_student', S.student_id)}	A class map consisting of table expression referencing 2 tables and an Uri pattern that eventually is concatenation of 4 values: literal '*program_*', value of column *STUDENT.pname*, literal '*_student*' and value of column

	STUDENT.student_id (concatenation of the first 2 values is specified explicitly by | | operator).

An object property map is described by a table expression, containing exactly one source class map reference (a class map reference with *mark*=<*s*>) and exactly one target class map reference (*mark*=<*t*>) within the expression's declaration structure.

A class map reference mark <*s*> or <*t*> can be included into object property *p* map expression structure either with a preceding class map description, or without it. The inclusion of a class map description within an object property map's expression means defining in-place a new class map that the object property map is going to refer to as its source or target. The most common usage of the construct, however, is without the explicit class map description; in this case, the mark <*s*> (resp. <*t*>) refers to the single class map that is ascribed to the domain (resp. range) class of *p*. Some examples of object property descriptions:

`<s>, <t>;` `<s>.teacher_id = <t>.teacher_id`	This table expression follows requirement for object property map description about class map references <s> and <t>. Therefore, this is also an object property map's description.

`(`
`  STUDENT`
`  {uri=('Student',student_id)}`
`) <s>,`
`REGISTRATION R,`
`(`
`  COURSE {uri=('Course', course_id)}`
`) <t>;`
`<s>.student_id =R.student.id`
`  AND R.course_id=<s>.course_id`
	This object property map's description contain explicitly written (inline) class map that is referenced by <s>. In metamodel it corresponds to explicit link from the *ClassMapRef* to *ClassMap*.
	Similarly, a reference to the target class map is stated inline.

If the first of the above expressions were attached to the OWL object property *teaches* (Teacher teaches Course) then <s> would mean a reference to the sole class map that is attached to the *Teacher* class (domain class for the property).

A data property map is described by a table expression that is required to contain a single <*s*>-marked reference to the source class map, followed by a value expression attached to the table expression using a dot notation and further on by an optional datatype specification preceded by the string '^^'. Value expression is built in accordance to the abstract syntax of expression and filter metamodel of **Figure 16**)

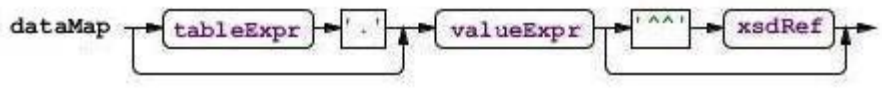


Figure 19. RDB2OWL Raw data property map syntax

Some examples of data property descriptions are:

`<s>.name`	A simple data property map description referencing source class map (not inline) by <s> and specifying *name* column

	for value expression
`<s>.name^^xsd:String`	Similarly as above with explicit xsd datatype added.
`( TEACHER {uri=('Teacher',teacher_id)} <s> ).name`	Inline defined source class map and *TEACHER.name* for value expression
`( COURSE {uri=('Teacher',teacher_id)} <s>, PROGRAM P; <s>.program_id=P.program_id ).('program: ' || P.name || ', course: ' || <s>.name )`	Expression, not merely column reference is used for value specification.
`( (TABLE1 {uri=('Something',table1_id)})<s>, TABLE2 T2, TABLE3 T3, TABLE4; <s>.table2_id=T2.table2_id AND T2.table3_id=T3.table3_id AND T3.table4_id=T4.table4_id ). ( <s>.name || ' ' || T2.name || ' ' || T3.name || ' ' || T4.name )`	This data property map's expression is for *farPath* property in far table linking example [2.3.2] where 4 tables are joined and data value is formed by concatenating name column value from all these tables. There is inline description for source class map and reference to it by the <s> mark.

What we said about inline class map definitions for reference marks <s> and <t> in object property map description can be applied also for data property map descriptions but with respect to mark <s> only. In the case when a table expression part for a data property map is just *<s>*, we allow omitting it together with the following dot symbol for a short form of mapping specification. The following expressions are equivalent:

- *<s>.name*
- *name*

Similarly, the declaration part (together with the following semicolon) may be omitted for an object property map, if it is just *<s>, <t>* therefore, the following expressions are equivalent either:

- *<s>, <t>; <s>.course_id = <t>.course_id*
- *<s>.course_id = <t>.course_id*

4.1.3 RDB2OWL Raw mapping specification usage

If an OWL class were mapped to a table context consisting of more that one table then class map's *TableExpression* instance would contain a separate *RefItem* instance of *TableRef* type for each table of the context. The tables are joined by SQL expression in *filter* attribute. There is some similarity with how in SQL statement several tables are introduced by optional aliases in FROM clause and joined in WHERE clause. Suppose we have a case when tables *TABLE_1, TABLE_2, ..., TABLE_n* are introduced with optional aliases *A1, A2, ..., An* and joined by expression *JOIN_EXP(A1, A2,..., An)* that references columns with prefix of aliases

or tables names or without prefix (if evaluation is unambiguous). Then in SQL we would write

```
SELECT …
FROM TABLE_1 A1, TABLE_2 A2, …, TABLE_n An
WHERE JOIN_EXPR(A1, A2, …, An)
```

But in RDB2OWL metamodel the tables are introduced and joined to OWL class *C* as shown in figure below followed by ClassMap description in concrete syntax

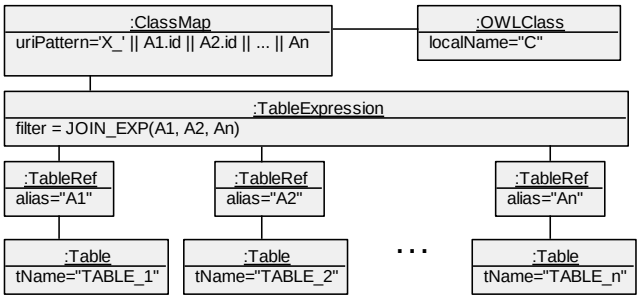


Figure 20. Class map linked to table context of many joined tables

```
TABLE_1 A1, TABLE_1 A1, …, TABLE_n An;
JOIN_EXPR(A1, A2, …, An) {uri=('X_' || A1.id || A2.id || …|| An.id)}
```

Figure 21 shows RDB2OWL metamodel instances that define all class maps for the mini-university example in abstract syntax (if not stated otherwise, examples further will be based on mini-university example). Note that there are two class maps, generating instances of the *PersonID* OWL class. Note also that the *Teacher* and *Course* classes have virtual class maps but their subclasses *Assistant*, *Professor*, *AssocProfessor* and *MandatoryCourse*, *OptionalCourse* have non-virtual class maps attached. It is because there is no need for instance triples for superclasses if they are generated for subclasses. The table expressions for subclasses show a typical mapping pattern for subclasses if their instances correspond to subset of all table rows-superclass and subclass map to the same table but the subclass has additional row filtering expression. For example, superclass *Course* and subclass *MandatoryCourse* are both mapped to table *COURSE* but filter=”required=1” is added for subclass.

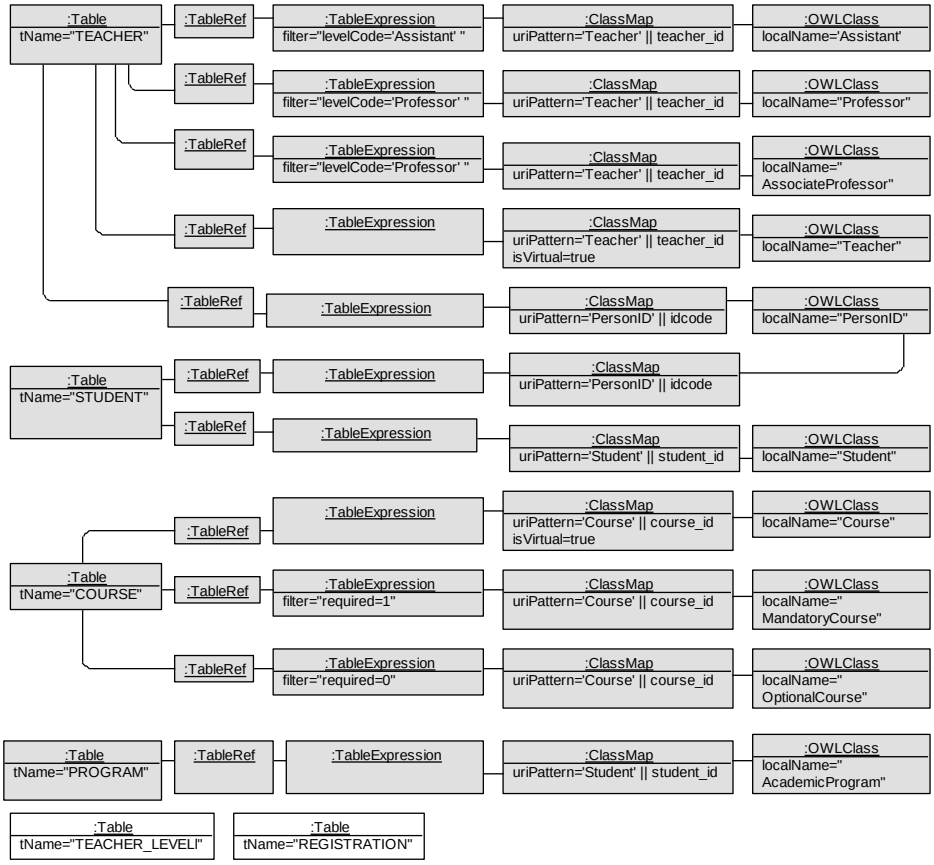


Figure 21. RDB2OWL Raw Mapping instances for Mini-university example: class maps

Below are shown three class map descriptions in concrete syntax for *Teacher*, *Course* and *MandatoryCourse* respectively (note the decoration "*!No*" for virtual class map) :

```
TEACHER; levelCode='Assistant' {uri=('Teacher', teacher_id)}
COURSE {uri=('Course, course_id)} !No
COURSE; required=1 {uri=('Course', course_id)}
```

The instance model shows that two tables are without any class map. In this case it an intension of the mapping's author but not always so. The RDB2OWL model can help to detect omissions by mistake. Other validation features are also possible, e.g. find un-mapped OWL classes or classes with unintended double class maps.

The **Figure 22** below shows RDB2OWL model instances for property maps. OWL classes *Teacher* and *Course* have class maps that are based on DB tables *TEACHER* and *COURSE*, respectively. The object property map for the property *teaches* has a table expression consisting of two class map references (*ClassMapRef*) with aliases <s> and <t> and that has *ref* links to the same class maps that are attached to the *Teacher* and *Course* classes respectively, that is, to the source and target class maps.

Observe that the mapping of the property *teaches* is implemented using virtual class maps to the *Teacher* and *Course* classes (the "real" instance generation in *Teacher* and *Course* classes has been specified for their subclasses). Similarly, data property map for the *courseName* property has a table expression with <s>-marked class map reference with *ref* link to the same class map that is attached to *Course* class (see **Figure 22** below).

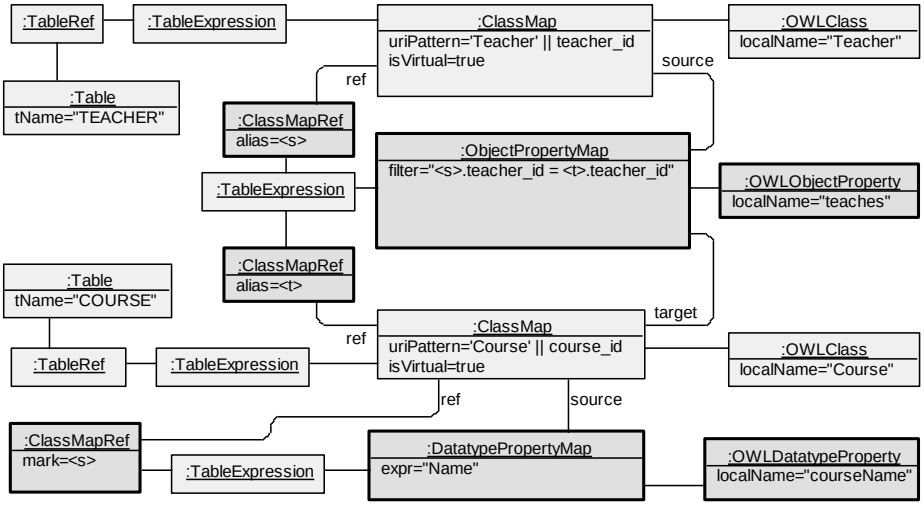


Figure 22. RDB2OWL Raw Mapping instances for Mini-university example: object and data property maps

Concrete syntax for ClassMap, ObjectPropertyMap, ClassMap and DatatypePropertyMap of **Figure 22**:

```
- TEACHER {uri=('Teacher', teacher_id)} !No
- <s>, <t>; <s>.teacher_id = <t>.teacher_id
- COURSE {uri=('Course', course_id)} !No
- <s>.Name
```

The "standard" solution to specification of the object property *takes* that is mapped to join of tables STUDENT and COURSE through intermediate REGISTRATION table is shown in **Figure 23** below. We notice that the subclass optimization feature of RDB2OWL Raw Plus (discussed in Section 4.3.1) would achieve a similar effect also, if the class maps referred from the *teaches* property map definition were not virtual. The virtual class maps allow achieving the needed triple set locally, without invoking the general subclass optimization principle.

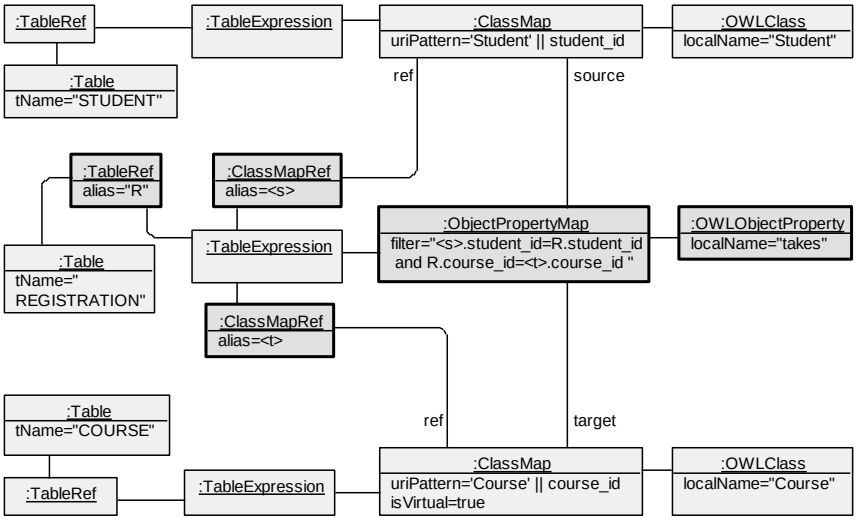


Figure 23. Mapping instances: property mapping through linked table

Concrete syntax for the two *classMaps* (referred by <s> and <t>) and ObjectPropertyMap of **Figure 23** is:

```
- STUDENT {uri=('Student', student_id)} !No
- COURSE {uri=('Course', course_id)} !No
- <s>, REGISTRATION R, <t>; <s>.student_id=R.student_id AND
    R.course_id=<t>.course_id
```

There is, however, an alternative object property mapping solution depicted in **Figure 24** for the *takes* property defined as follows. The *Student* and *Course* classes have attached class maps responsible for class instance generation. Standard solution would use these "real" class maps to generate subject *s* and object *o* parts for property's value triples *<s, ex:takes, o>*. The alternative way is to define two new virtual class maps of the *Student* and *Course* classes for the *s* and *t* generation and attach them to the object property map as source and target class maps. This way the subject *s* and object *o* is generated directly from *REGISTRATION* table that contains the *student_id* and *course_id* columns that are needed for URI generation. The down side of this solution is a need to re-specify the URI patterns for subject and object URI generation, however this possibility outlines the power of the virtual class maps.

The virtual class maps in the style of **Figure 24** are essential, if we want to define several RDB-to-RDF/OWL mappings, each of them responsible for a certain source database, and if we want to create some cross-database linking properties (e.g. on the basis of certain field value equality), where the mapping A cannot access the instance-generating class map that is defined within the mapping B.

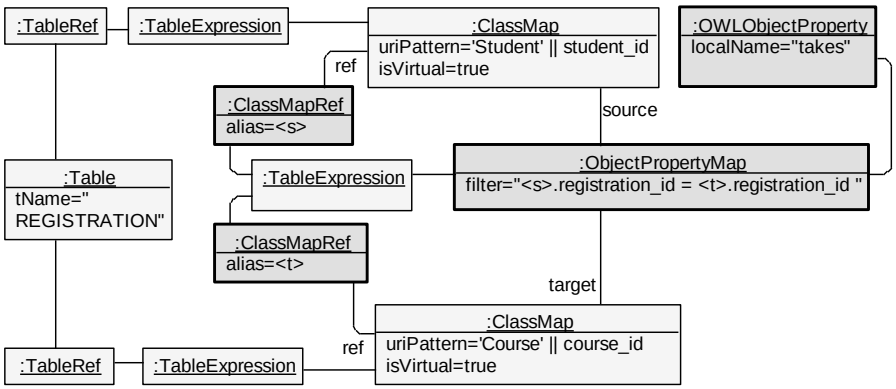


Figure 24. Linking through table: virtual class maps

Concrete syntax for *ObjectPropertyMap* with two inline defined classMaps of **Figure 24**:

```
(REGISTRATION {uri=('Student', student_id)} !No) <s>,
(REGISTRATION {uri=('Course', course_id)} !No) <t>;
<s>.registration_id=<t>.registration_id
```

Figure 25 shows how 3 additional tables are joined to data property map through its table expression. The *farName* data property from far table linking example (see Section 2.3.2) have a table context comprising 4 tables- one from the source class map's table expression and 3 additional. The *filter* attribute contains a table joining expression and the *expr* attribute contains expression- a single column from the 4-th table *TABLE4*.

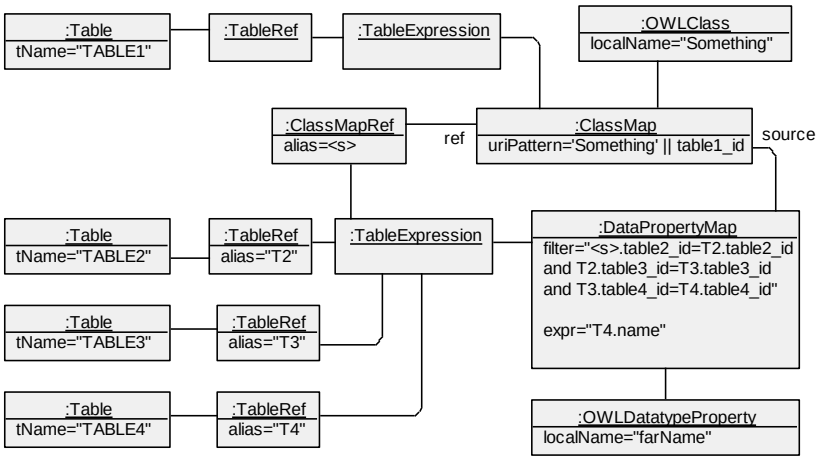


Figure 25. RDB2OWL Raw Mapping instances for far table linking example

4.1.4 RDB2OWL Raw annotations for Mini-University example

For a complete example, we show RDB-to-OWL mapping specifications as annotations in the target ontology according to RDB2OWL Raw language. The target ontology and source database for mini-university example is described in Section 2.3.1.

All mapping definitions are shown in **Figure 26** below. We use here a custom extension of UML-style OWL Graphic Notation editor OWLGrEd [14] depicting *DBExpr* annotations in the form '*{DB: <annotation_text>}*' to show graphically the ontology together with the annotations. The class and object property annotations in the example are shown in italics, while the data property annotations use plain text. The naming convention is used to writing database table names in uppercase letters in order to distinguish class names from table names, for example, Student class from STUDENT table.

The RDB2OWL raw mapping format reveals the structure of the information that needs to be specified in order to define the mapping. The syntactic presentation of the mapping, however, is less than satisfactory, especially for not 1:1 correspondence cases between the RDB and ontology structure. For example, the *personID* property has explicit class map definitions included within each of property maps description. The Uri pattern for Teacher class map is repeated for all three class maps of subclasses (the same for Course class). The foreign key and primary key columns are explicitly written to specify table joining for object property maps, e.g., *<s>.teacher_id=<t>.teacher_id*. It means long typing especially if more than two tables joined, e.g. for object property *takes*.

These and other issues, as well as more compact and better structure revealing forms for class and property map definitions are handled in RDB2OWL Core notation

in Section [4.2] where some information can be omitted that can be deduced from database and ontology structure.

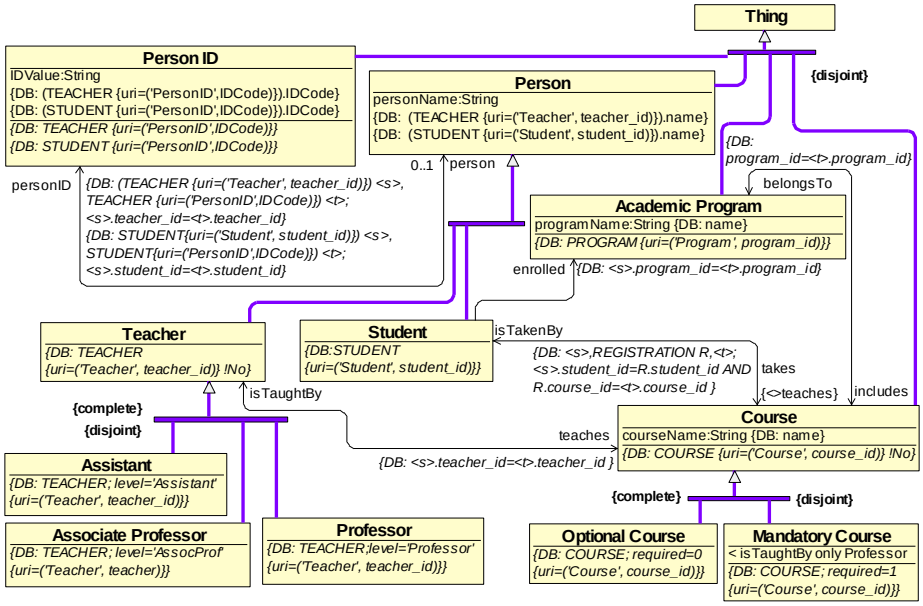


Figure 26. Annotated mini-University ontology using RDB2OWL Raw model

4.1.5 RDB2OWL Raw annotations for Far link example

On the far table linking example (described in section [2.3.2]) we show how to define data property maps in case when value expressions for properties involve table column from other tables- not only from the one the class map is based on.

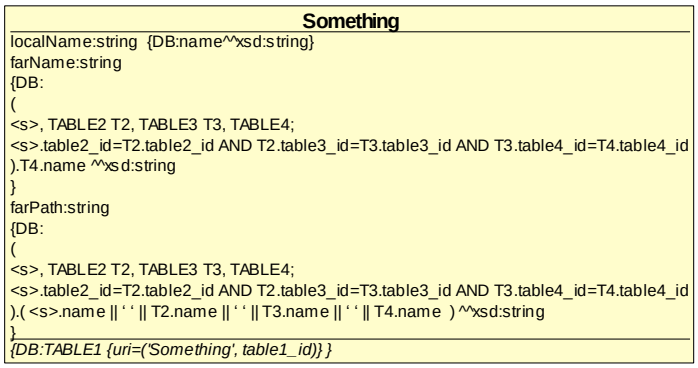


Figure 27. Annotated far table linking ontology using RDB2OWL Raw language

The ontology consists of one OWL class and 3 data properties. The *Something* class is mapped to the TABLE1 table by a class map. Value for the *farName* property

is taken from the *name* column of the *TABLE4* table that is linked to the *TABLE1* table by a long chain of linked tables: *TABLE1* →*TABLE2* →*TABLE3* →*TABLE4* with links defined on equality of respective columns. Value for the *farPaths* property is formed by concatenation of the values in the *name* field from all linked tables.

4.1.6 RDB2OWL Raw annotations for Genealogy example

The Genealogy example (see Section 2.3.3) has a mapping peculiarity: there is no table corresponding to the *Gender* class to define a class map on. One way to solve such a mapping problem is to define new table or view as an auxiliary database schema object that would return required data for the *Gender* class (see Section 4.3.2 about auxiliary database objects in RDB2OWL Core Plus language):

```
CREATE TABLE gender (gender_name VARCHAR(10));
INSERT INTO gender (gender_name) VALUES ('female');
INSERT INTO gender (gender_name) VALUES ('male');
```

We show how to define a class map without referencing any database table and not introducing new database schema objects. Note that a table expression of class map can be empty, that is, having no reference items at all. If a class map without table referenced have Uri pattern definition as literal value then such class map specifies one concrete triple generation (for *rdf:type* predicate). For the *Gender* class there have to be defined two such "literal" class maps: one for the *female* and one for the *male* value. The annotated ontology is shown in **Figure 28** below.

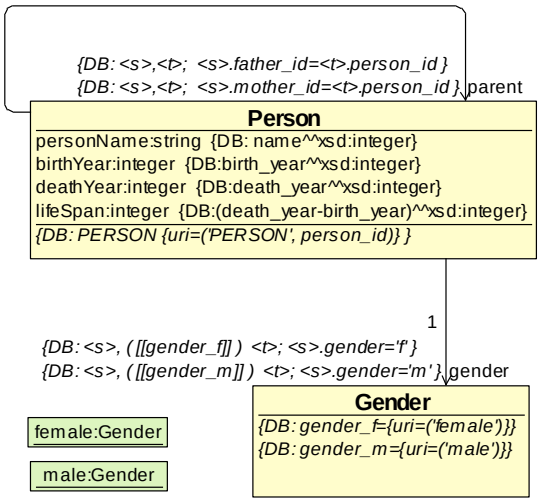


Figure 28. Annotated genealogy ontology using RDB2OWL Raw language

The two "literal value" class maps to the *Gender* class are named by *gender_f* and *gender_m* and are referred from the object property map for the *gender* property. A row from the *PERSON* table satisfying condition *PERSON.gender='f'* is used for the value *female* formation for the *gender* property (similarly condition *PERSON.gender='m'* for the value *male*). Note that <s> refers to the *PERSON* table (the table of the table expression of the source class map of the *gender* property).

Note also the arithmetic expression for the data property map of the *lifeSpan* property. The *parent* property has the same *Person* class for the domain and range. Therefore, the same *PERSON* table is used for source and target class maps. By using different aliases (<s> and <t>) resolves which one of the two linked *PERSON* tables to refer. The *parent* property needs two property maps because database has two foreign-to-primary keys from the *PERSON* table to itself: one for the mother and the other for the father link specification.

4.1.7 RDB2OWL Raw Mapping Semantics

The RDB2OWL mapping semantic is about how to generate RDF triples corresponding to the target ontology from mapping expressions (class maps and property maps) and source database data. It should be considered for tool developers that process RDB2OWL mapping annotations. The total semantic reduces on semantic of the main language constructs. In case of nested constructs, the semantic of higher level construct is defined by semantic the lover level construct. For example, the semantic of a class map is determined by semantic of the included table expression.

If class map reference mark <*s*> resp. <*t*> in property map *pm* for property *p* is spelled without inline defined class map then by simple transformation a referenced class map can be copied it in:

(a) lookup domain resp. range class *c* for property *p* from ontology structure information,
(b) find the only class map expression *m* that is ascribed to class *p,*
(c) copy *m* into property map *pm* behind the <s> resp. <t> mark:
"… <s>… → … (m) <s> …" or "… <t>… → … (m) <t> …"

For example, the transformation for object property map, that is assigned to the *takes* property, would copy the class map definitions from the *Student* and *Course* classes:

```
<s>, REGISTRATION r, <t>; <s>.student_id=r.student_id and
r.course_id=<s>.course_id
→
(STUDENT {uri=('Student', student_id)} ) <s>,
REGISTRATION r,
(COURSE {uri=('Course', course_id)} ) <t> ;
<s>.student_id=r.student_id and r.course_id=<s>.course_id
```

Therefore we can assume without losing generality that the source (resp. target) class map descriptions for property maps are textually included behind the corresponding <*s*> (resp. <*t*>) marks and treat the marks <*s*> and <*t*> themselves as ordinary aliases.

The context *C(t)* for the table expression *t* is built inductively following *t* structure:

(a) if *t* is a reference to a table with name *tName*, then *C(t)* consists of expressions *tName.cName* for *cName* ranging over all *t* column names;
(b) if *t* is of the form *t' a* for table or a table expression *t'* and an alias *a*, then *C(t) = {a.x | x ∈ C(t')}*;

(c) if t consists of items t_1, …, t_n with no aliases specified then
$C(t) = C(t_1) \cup C(t_2) \cup ... \cup C(t_n)$; if for some t_i there is an alias specified, the rule (b) is to be applied on this item before (c).

We require that the fully qualified names in the table expression context be distinct; if that is not the case, the table expression is not well formed.

Given a table expression t for a property map m, we denote by *src(t)* (resp. *trg(t)*) the column prefix within *C(t)* that ends in <*s*> (resp. <*t*>); the requirements on t structure ensure that *src(t)* (resp. *trg(t)*) is uniquely defined. Note that, for instance, for a table expression *t=(A <s>) E1, B <t>* we have *src(t)=E1.<s>* and *trg(t)=<t>*.

For a value expression x and a string a we define the a-lifted form of x by replacing every column reference t within the x structure by $a.t$.

The semantics *R(t)* of a table expression t on the source database $\boldsymbol{S}$ is defined as a set of rows with columns corresponding to the table expression context *C(t)* the following way. If there is no filter expression specified as part of t, then

(a) if t is a table, then *R(t)* consists of all its rows
(b) if t is of the form *t' a* for a table expression *t'* and an alias a, then *R(t)* is obtained by renaming *R(t')* columns via adding the prefix '*a.*'
(c) if t consists of items t_1, …, t_n with no aliases involved then *R(t)* is formed by taking all row combinations from the row sets $R(t_1)$, ..., $R(t_n)$.

If, however, there is a filter specified as part of t, only the rows that satisfy the filter are retained in *R(t)*, as obtained above.

For every row $r \in R(t)$ there is defined notion of value expression evaluation: the column expressions are looked up within the row; the constants and standard operators have their usual meaning. Let *concat* be a function concatenating all its arguments.

Given the source database schema $\boldsymbol{S}$, the corresponding RDF triples are defined for each class map and property map separately.

For a class map or property map m let t be the table expression contained in m and let e be the OWL entity (OWL class or OWL property) that m is ascribed to (we consider only class maps ascribed to OWL classes here). Let r be the *baseURI* specified for the target OWL ontology. In order to form the RDF triples that correspond to $\boldsymbol{S}$, we form in each case the row set *R(t)* by evaluating t on $\boldsymbol{S}$. For each row in *R(t)* we then proceed, as follows:

- if m is a class map, evaluate the expression contained in m's *uriPattern* attribute obtaining a string value v; then form the RDF triple <*concat(r,v),* 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type', *e.entityURI*>;
- if m is an object property map, evaluate the *src(t)*-lifted form of m's source class map *uriPattern* to obtain u and the *trg(t)*-lifted form of m's target class map *uriPattern* to obtain v, form the triple <*concat(r,u), e.entityURI, concat(r,v)*>;
- if m is a data property map, let d be the value of m's *expr* attribute evaluation; let s be obtained by evaluating the *src(t)*-lifted form of m's source class map *uriPattern* value. Further on we find *dt*: XSD datatype corresponding to d the following way:
    - if there is an XSD datatype specified within m, take this datatype
    - if the XSD datatype can be found as a default XSD datatype for d's SQL datatype, take this datatype

- if the XSD datatype has been specified as *e.range*, take this datatype
- if none of the above applies, take *dt* to have *typeName='xsd:String'*.

The resulting triple is <*concat(r,s),e.entityURI,concat(d,'^^',dt.typeName)*>.

For example, let us see what triples are created for class map *"COURSE; required=1 {uri=('Course',course_id)}"* that is attached to class *MandatoryCourse* (see **Figure 26**). The table exprsession included is t=*"COURSE; required=1"* and *R(t)* is set of *COURSE* table rows satisfying condition "*required=1*":

Rows(course_id, name) = {(2,'Semantic Web') , (3,'Computer Networks') }

uriPattern evaluation v= {'Course2', 'Course3'}

If baseURI of target ontology *r='http://lumii.lv/ex#'* then we obtain RDF triples calculating <*concat(r,v),* 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type', *e.entityURI*>:

Subject	**Predicate**	**Object**
http://lumii.lv/ex#Course2	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://lumii.lv/ex#MandatoryCourse
http://lumii.lv/ex#Course3	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://lumii.lv/ex#MandatoryCourse

4.2 RDB2OWL Core

RDB2OWL core language augments RDB2OWL raw metamodel (Figure 15, Figure 16) with a list of additional constructs to enhance expressiveness and allow for more concise and readable expression forms.

The RDB2OWL core language constructs are summarized in metamodel in Figure 29. The core language constructs are semantically explained via their translation into the augmented RDB2OWL raw language. The principal novelties here are:

- possibility to add top rows constraint to reference items of table expressions;
- a more refined table expression structure (each table expression reference list item now may be expressed as list-like navigation item and link structure);
- the primary key and foreign key information within RDB MM; this allows:
  (i) introducing default entity URI patterns on the basis of RDB schema structure: the default Uri pattern for a class map, whose table expression is a single table *X* having a sole primary key column *P,* is defined as *uri('X',P)*; and
  (ii) avoiding the necessity to specify column names in linked table conditions, if these correspond to a unanimously clear table key information;
- the naming of class maps (*defName* attribute), together with a possibility to refer within a table expression item (a *NamedRef* instance) either to such a defined class map, or to the sole un-named class map defined for a certain

OWL class *c*. A named reference syntactically is represented as '*[[R]]*', where '*R*' is either the name of an owl class, or the defined name of a class map.

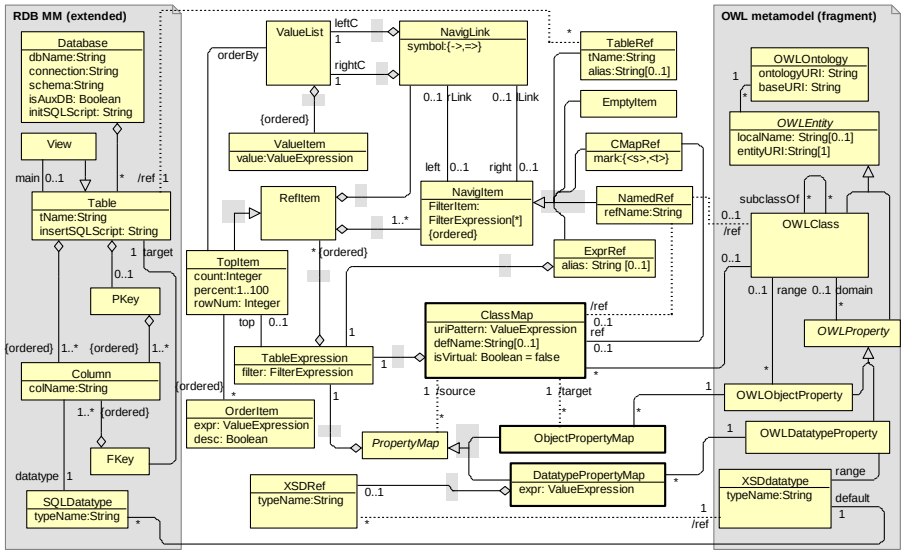


Figure 29. RDB2OWL Core metamodel

A reference item (*RefItem* instance) can be labeled a *top*-constrained element of a table expression. The semantics of such an element *el*, if designated for a table expression *t,* is reflected in the row set *R(t)* construction for *t* in that only the specified rows (one or more top rows or specified percent of top rows with respect to defined order) for columns corresponding to *el* is included into *R(t)* for a combination of values in other *t* columns. One can specify also concrete row, e.g., the third one (for example, in sports domain this could make sense for information about bronze medal position). Syntactically after reference item expression, we write top rows specification.

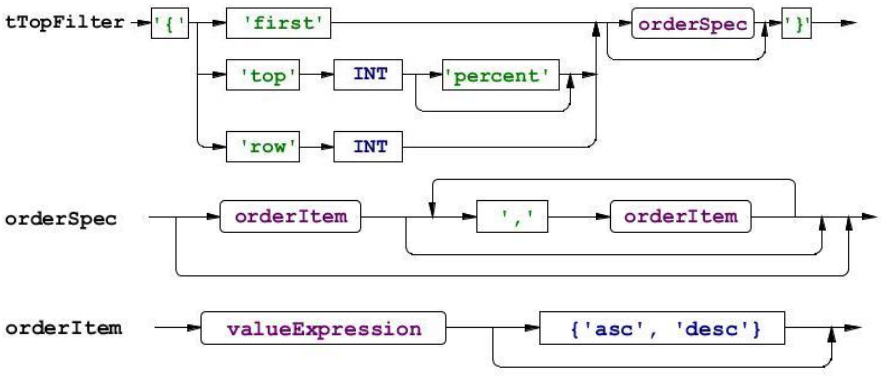


Figure 30. Top rows specification grammar

Some examples:

`(Student <s>, Program <t> {top 1 PName asc}; PName >'')`	denote all students and the first program with non-empty name for each of them
`PERSON father, PERSON child {row 3 birth_year desc}; father.father_id=child.father_id AND child.gender='male'`	denote all fathers and their third oldest sons
`PERSON {top 50 percent surname asc, name asc}`	the top half of all persons ordered by surname and then by name

The principal building blocks of the RDB2OWL Core metamodel, as in the Raw version of the model, are class maps and property maps that are based on table expressions, consisting of reference items. A reference item can contain a singleton list of navigation items thus subsuming the reference item structure of Raw metamodel. The navigation item and navigation link structure of a reference item allows to encode in simple table expressions several filter conditions as equality between columns in two adjacent navigation items.

For instance,

Student[Program_ID]->[Program_ID]XProgram

is a reference item presentation in a navigation item and link form, where *Student* and *Program* are navigation items and *[Program_ID]->[Program_ID]* is a navigation link with *Program_ID* being its left and right value lists (and the sole value items within these lists) respectively. The presented navigation item and link structure can be translated into an equivalent table expression *Student E1, Program E2; E1.Program_ID=E2. Program_ID*, where *E1* and *E2* are unique system generated aliases introduced to avoid potential table name conflicts. The navigation structures can form also longer chains as, for example, the following:

`<s>[StudentID]->[Student_ID]REGISTRATION[Course_ID]->[Course_ID]<t>.`

We allow an empty navigation item only in linked navigation structures (the ones containing link symbol -> or => adjacent to the empty item). In this case the empty item is shorthand for a class map reference <*s*> or <*t*>, and it is replaced by the reference during the expression's short form unwinding, the rules for the short form unwinding are explicitly formulated later on page 77 as a four-step process (steps 1. .. 4).

For a navigation link *A[F1]->[F2]B* between two single table items *A* and *B* (possibly included in expressions with filters and aliases, or being class map references or named references to single table class maps) the table column list *[F2]* may be omitted, if it corresponds to the primary key of *B*. Furthermore, *[F1]* may be omitted, as well, if there is a single foreign-to-primary key reference in the source RDB from *A* to *B* and it is based on the equality of the foreign key columns *F1* to primary key columns *F2*. For the case when *A[F1]->[F2]B* is a primary-to-foreign key relation, we allow omitting both *[F1]* and *[F2]*, in this case presenting the expression as *A=>B*. Below are longer and shorter navigation structure expression forms for the property *takes*:

`<s>[StudentID]->[Student_ID] REGISTRATION`	Class map references <s> and <t> actually introduce table expressions defined in them:

`[Course_ID]->[Course_ID]<t>`	STUDENT and COURSE respectively. These tables are joined with REGISTRATION on equality of columns *student_id* and *course_id* respectively
`<s> => REGISTRATION -> <t>`	Omitted primary key columns of the *STUDENT* and *COURSE* tables. Omitted also the *REGISTRATION*-to-*STUDENT* foreign key column and *REGISTRATION*-to-*COURSE* foreign key column

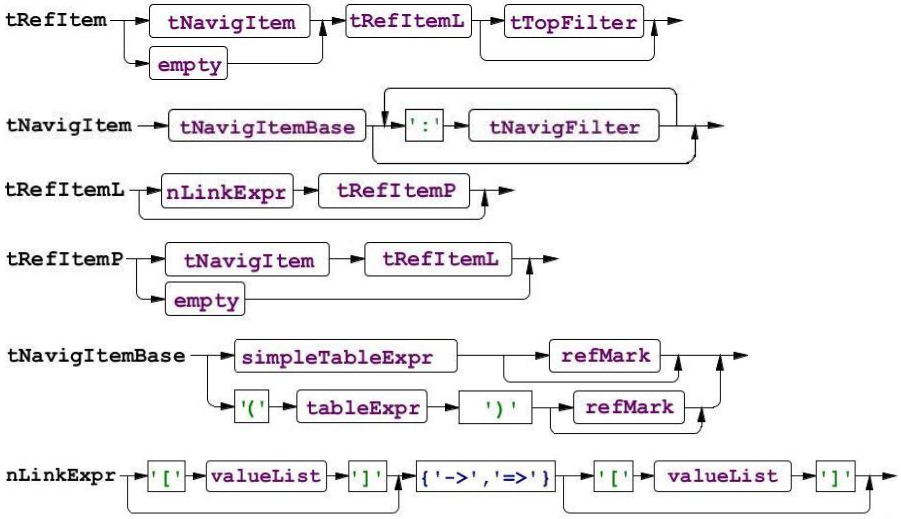


Figure 31. Grammar for navigation link expressions

The grammar for navigation links is a little complicate in order to deal with empty items (omitted navigation items for <s> or <t> class map references). The second branch of *tNavigItemBase* essentially resolves in recursion: *tNavigItemBase*→*tableExpr*→ … → *tRefItem*→*tNavigItem*→ *tNavigItemBase*. This allows coding nested table expressions, for example:

```
( STUDENT=>[student_id]REGISTRATION ) SR [SR.course_id]->COURSE
```

Longer chain of linked tables *t1->t2->t3->...->tn* can be established by recursion:

tNavigItem→ *tRefItemL*

→ (*nLinkExpr* →*tRefItemP* →*tNavigItem* →*tRefItemL*) (for table *t1*)

→ (*nLinkExpr* →*tRefItemP* →*tNavigItem* →*tRefItemL*) (for table *t2*)

...

→ (*nLinkExpr* →*tRefItemP* →*tNavigItem* →*tRefItemL*) (for table *tn*)

As an example, we show a parsing tree (generated in ALTLRWorks tool [74]) for simple two-table link expression:

```
TEACHER[teacher_id]=>COURSE
```

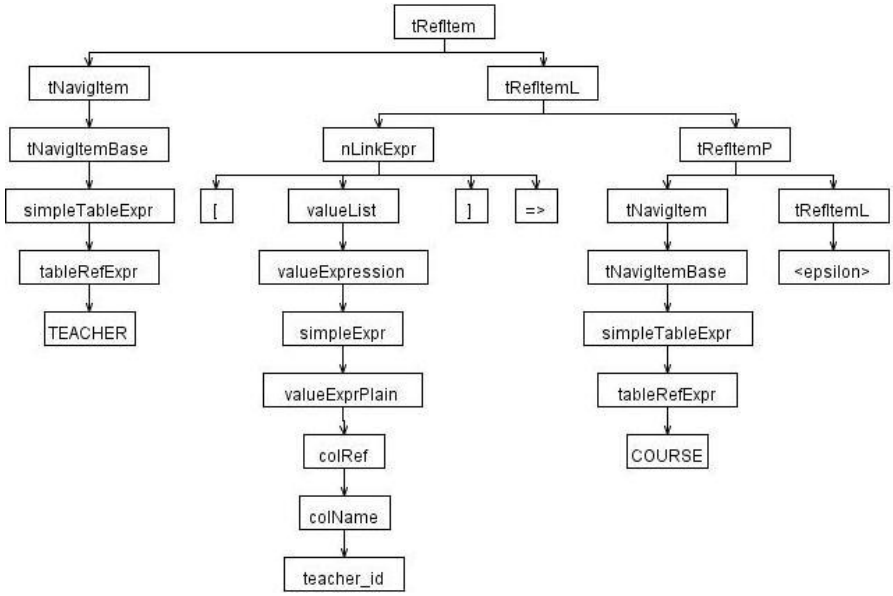


Figure 32. Parse tree for navigation link of two tables

A class map definition can be assigned to a variable and referred by variable name (*defName*). Reference to defined class map or to the sole class map defined for given class is the same. If reference ([[…]]) uses name but no variable name exists with such a name then the name is interpreted as OWL class name whose class map is referenced. Class map reference by class names is useful in situations when one needs to specify class map for subclass and already defined class map for superclass can be re-used thus allowing not writing duplicate class map expressions. For example, class map for *professor* class can reuse class map for *Teacher* class and add additional details, e.g., filter conditions (the same for all other subclasses of *Teacher* class)

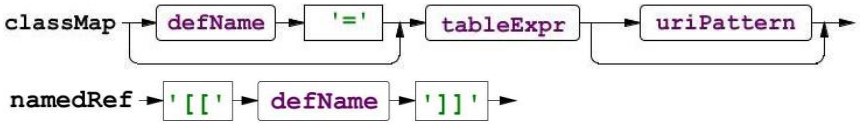


Figure 33. Defined names for class maps and references by name

Class map for the *PersonID* class: `T=TEACHER {uri=('PersonID',IDCode)}`	Assigns a name *T* do the defined class map

Class map for the *Teacher* class:
`TEACHER`
`{uri=('Teacher',teacher_id)}`	

Object property map for the *personID* property: `[[Teacher]][teacher_id]->[[T]]`	If no defined class map name exist with name *Teacher* then *[[Teacher]]* is reference to the sole class map defined for class Teacher; *[[T]]* is

	reference to the defined class map by name *T*
Class map for *Professor* class: `[[Teacher]]; level='Aprofessor'`	Reuse of class map defined for *Teacher* class

The table expressions corresponding to the RDB2OWL Core metamodel are transformed into the (augmented) RDB2OWL Raw format via the following steps:

A. Unwinding of the short form of table expressions (insertion of <*s*> and <*t*> class map references, where appropriate) for data and object properties, following the rules explained below in steps 1. .. 4.
B. Insertion of explicit Uri pattern definitions for class maps, in place of default Uri expressions (including both class maps ascribed to classes and defined inline).
C. Replacing named references by their referred class maps defined inline.
D. Insertion of explicit column information definition in navigation links.
E. Converting the navigation expression structure into reference item structure.

For a table expression reference list structure consisting of expression and navigation items we define its *leftmost* (resp. *rightmost*) item by recursively choosing expression's leftmost (resp. rightmost) reference or navigation item, until an item that is an empty item, a table reference, a class map reference (with or without an explicit class map specification), or a named reference, is found.

The rules for unwinding the short form of a table expression *e* corresponding to an object property map are, as follows:

1. if *e*'s reference item list is empty define this list to be <*s*>,<*t*>;
2. if the leftmost (resp. rightmost) item is an empty item, replace this item by <*s*> (resp. <*t*>); e.g. any of '<*s*>->', '-><*t*>' and '->' is replaced by '<*s*>-><*t*>' and '*(Person <s>)*->' is replaced by '*(Person <s>)*-><*t*>';
3. if <*s*> (resp. <*t*>) is not present in *e* reference list structure and is explicitly referenced from the filter expression, add <*s*> (resp. <*t*>) as a new leftmost (resp. rightmost) reference item;
   for instance '*Registration;<s>. Student_ID =Student_ID*' is replaced by '*<s>, Registration;<s>. Student_ID = Student_ID*';
4. if <*s*> (resp. <*t*>) is not present in *e* reference list structure, add <*s*> (resp. <*t*>) as an alias for the *e*'s leftmost (resp. rightmost) item; for instance '*Person, Program*' is replaced by '*Person <s>, Program <t>*'.

Similar rules (in the part regarding <*s*>) apply also for unwinding of the short form of table expressions involved in data property map definitions.

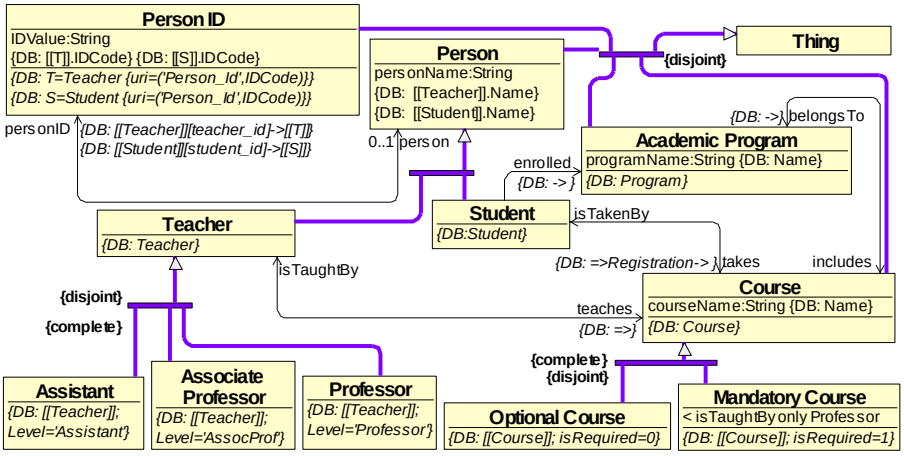


Figure 34. Annotated mini-University ontology using RDB2OWL Core model

Figure 34 shows annotated mini-university ontology of **Figure 3**, this time being re-worked in accordance to the core language constructs. Note the simple '->' annotations for object properties *enrolled* and *belongsTo*, as well as '=>' for *teaches*. We note also the use of named references expressed as *[[Teacher]]* (referring to the class map defined for the *Teacher* class) and *[[S]]* and *[[T]]* referring to the class maps defined as named class maps for the *PersonID* class.

We demonstrate the steps A .. E, as defined above, on the object property map expression *[[Teacher]][teacher_id]->[[T]]* of object property *personID* in **Figure 34**.

A. *([[Teacher]] <s>)[teacher_id]->([[T]] <t>)* (unwinding rule 4. used).
B. Applies to converting *Teacher* into *Teacher {uri=('Teacher', teacher_id)}* within *Teacher* class (*[[T]]* already denotes an expression with *uri* specified).
C. *((Teacher {uri=('Teacher', teacher_id)}) <s>) [teacher_id] -> ((Teacher {uri=('PersonID',IDCode)}) <t>)*;
D. *((Teacher {uri=('Teacher', teacher_id)}) <s>) [teacher_id] -> [teacher_id] ((Teacher {uri=('PersonID',IDCode)}) <t>)*
E. *((Teacher {uri=('Teacher', teacher_id)}) <s>) E1, ((Teacher {uri=('PersonID',IDCode)}) <t>) E2; E1.(<s>. teacher_id)=E2.(<t>. teacher_id)* (the filter can be written also as *E1. teacher_id =E2. teacher_id* or *<s>. teacher_id =<t>. teacher_id*).

Figure 34 shows that mapping definition for a simple (still not completely straightforward) mapping case of mini-University can be done in RDB2OWL Core language in a very compact, yet intuitive way.

Figure 35 below shows annotated far table link example ontology consisting of one OWL class and three data properties. Note the conciseness of filter expressions compared with RDB2OWL raw expressions for the same example in **Figure 27**. We

suppose that tables *TABLE1, TABLE2, TABLE3 and TABLE4* are joined by chain of foreign-to-primary keys.

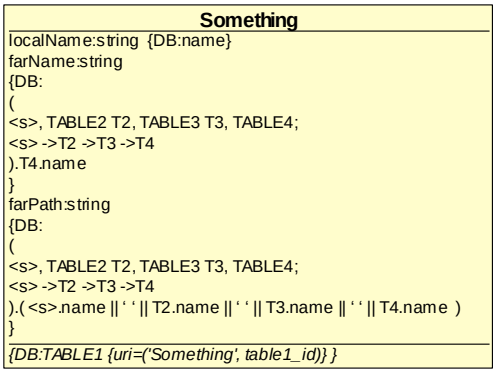


Figure 35. Annotated far link example ontology using RDB2OWL Core model

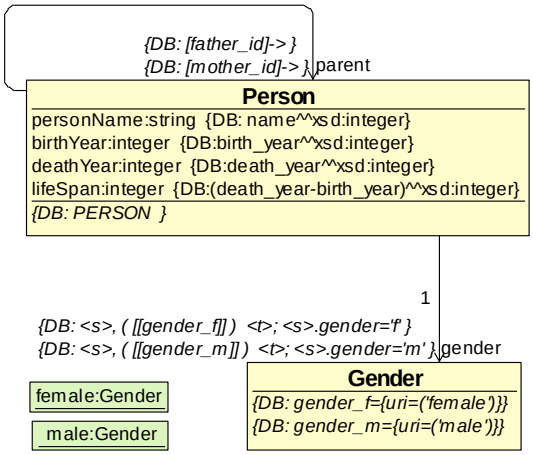


Figure 36. Annotated genealogy ontology using RDB2OWL Core model

Figure 36 shows RDB2OWL Core mappings expressed in a shorter way than in RDB2OWL Raw of **Figure 28**: navigation link -> without explicit right column written. Foreign key columns *farther_id* and *mother_id* need to be written explicitly because there are two foreign-to-primary keys from table *PERSON* to itself. Note also that class map for *PERSON* class omits default Uri pattern (concatenation of table name and primary key column value).

4.3 RDB2OWL Core Plus

RDB2OWL Core mapping language can be sufficient for simple applications but real life practical use cases show the need for various extensions while keeping the mappings compact and intuitive. The extensions described in this section are related to their practical use in a case study of using RDB2OWL approach to migration into RDF format of 6 Latvian medical registries [11, 12].

4.3.1 Multiclass Conceptualization

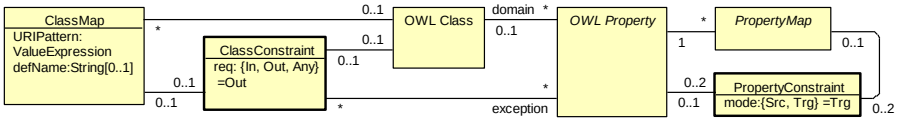


Figure 37. Class and property constraint metamodel

The meta-models of OWL ontology and RDB schema differ in that the former foresees a subclass relation, while the latter does not. We enhance RDB2OWL mapping language to deal with use cases where this difference is exploited. A *multiclass conceptualization* is a mapping pattern where one database table *T* is mapped to several ontology classes *C1*, *C2*,…,*Cn* each one reflecting some subset of *T* columns as the class' properties. In a standard way one would map each of *Ci* to the table *T* and would add to the respective class maps for *Ci* filtering expressions stating that only those rows of *T* correspond to *Ci* instances where at least one of the columns from the column set corresponding to the class' property column set has been filled. Mappings of Latvian Medicine registries contain such patterns where tables with several hundred columns are split into subsets of 20-30 columns. Filtering conditions for these mappings are lengthy and difficult to write and read.

Table 15. Multiclass conceptualization

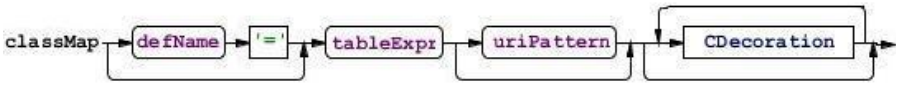
Table T	OWL Class	OWL Property	Lengthy filtering conditions to be avoided in the class map
col_a1	ClassA	propA1	`col_a1 IS NOT NULL`
col_a2		propA2	`OR col_a2 IS NOT NULL`
...		...	`...`
col_a60		propA60	`OR col_a60 IS NOT NULL`
col_b1	ClassB	propB1	`col_b1 IS NOT NULL`
col_b2		propB2	`OR col_b2 IS NOT NULL`
...		...	`...`
col_b40		propB40	`OR col_b40 IS NOT NULL`

To handle issue we introduce the *ClassConstraint* class whose instances specify requirement: a RDF triple *<x,'rdf:type',o>* can exist in the target triple set only if a triple *<x,p,y>* exists for some property *p* with domain *o* and some resource *y* (in the

terms of the last paragraph *x* would be an individual corresponding to a row in table *T* and *o* would be some class *Ci*).

If a class constraint is attached to an OWL class *c* it means that all generated instances *x* of *c* (the triples <*x,'rdf:type',c*>) should be checked for existence of property *p* instances for incoming (*p.range=o*), outgoing (*p.domain=o*, the default) or any (incoming or outgoing) properties. If a class constraint is attached to a class map, then it applies only to class instances that are created in accordance to this class map. The *exception* link from a class constraint specifies what properties are not to be looked at when determining the property existence. In Latvian Medical registries, we have used class level constraints for 54 out of 172 OWL classes with 514 out of 814 OWL data properties belonging to these constrained classes.

Syntactically we add class map decoration constant to express the described class constraint:




```
CDecoration : '?' | '?Out' | '?In' | '!No' | '!SubClean';
```

Figure 38. class map syntax with decoration

Meaning of class map decoration constants are: *?Out*- check outgoing properties, *?In*- check incoming properties, *?*- incoming and outgoing. Decoration *!No* specifies virtual class maps as described in section 4.1.2 (see Figure 18). *!SubClean* means requirement to delete generated instance triple for class if instance triple exist for subclass: delete <*x,'rdf:type',c*> if there exists generated triple <*x,'rdf:type',d*> for some subclass *d* of class *c*.

A class map expression for the *ClassA* class without multiclass conceptualization for the above case would be:

```
T; col_a1 IS NOT NULL AND col_a2 IS NOT NULL AND … col_a60 IS NOT
NULL
```

By using multiclass conceptualization to check the outgoing properties the class map can be written in a much shorter way omitting filtering condition at all:

```
T ?Out
```

A *PropertyConstraint* instance attached to a property *p* means requirement: check if for the subject *s* (*mode=Src*) or object *t* (*mode=Trg*) from a <*s,p,t*> triple there exists the triple <*s,*'rdf:type'*, p.domain* > (or <*t,*'rdf:type'*, p.range*>) generated by the mapping (if the check fails, delete the <*s,p,t*> triple). The checks associated with property constraints are to be applied after the class constraint resolution. Note that the property constraints with *mode=Trg* apply only for object properties. In Latvian Medicine registries there is a case of sugar diabetes mapping where property constraint appear essential in conjunction with class constraint use.

The class and property constraints are part of the mapping definition, not part of the target OWL ontology. The meaning of these constraints is fully "closed world": delete the triple, if the additional context is not created by the mapping.

4.3.2 Auxiliary Database Objects

There are cases when direct mapping between source RDB and the target ontology is not possible or requires complex expressions involving manual SQL scripts. Additional databases and its tables can be introduced for the mapping purpose. There can be multiple *Database* class instances in RDB2OWL core metamodel (see **Figure 29**). If the value of *isAuxDB* attribute is *true* then the database is auxiliary otherwise, it is a source database. A SQL script can be executed (attribute *initSQLScript*) to create necessary schema objects in auxiliary database and populate the tables with the needed data (attribute *insertSQLScript* of *Table* class). The definition and data of new auxiliary schema objects are considered part of the mapping specification.

The auxiliary tables and views can be used to simplify mapping presentation.

Another, more fundamental, usage context for auxiliary tables is ontology class or property that would naturally correspond to a database schema object that does not exist in the source RDB schema. A typical case of this category is a non-existing classifier table, which naturally appears in the ontological (conceptual) design of the data. In **Figure 39**, the OWL class *PrescribedTreatment* is based on database table *PatientData*. The *PatientData* table has "similar" binary attributes indicating that certain treatments on the patient have been performed. In the ontological modeling, one would introduce a single *diabetesTreatment* property to reflect all the "similar" fields from the *PatientData* table, the different fields being distinguished by different instances of the *DiabetesTreatment* class. The instances within the *DiabetesTreament* class may be specified either by directly entering them into the target ontology, or one could create an extra classifier table within an auxiliary database (a *TreatmentCategory* table in the example) that can be seen as a source for *DiabetesTreatment* instances.

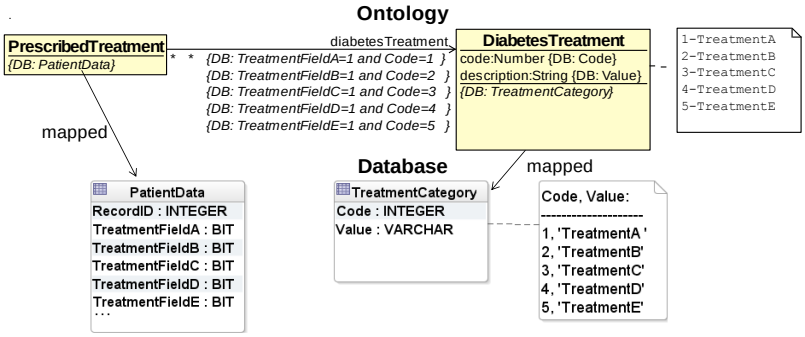


Figure 39. Ontology and Database fragment for Diabetes Treatment modeling

We add more detail to RDB2OWL metamodel by splitting *Database* class of **Figure 29** into two classes *Database* and *DBRef* separating external database connection information from data about interior references to the database and splitting *TableRef* into *TableRefExpr* and *TableRef* to separate navigation item expression with its alias and DB table reference by means of *dbAlias*.

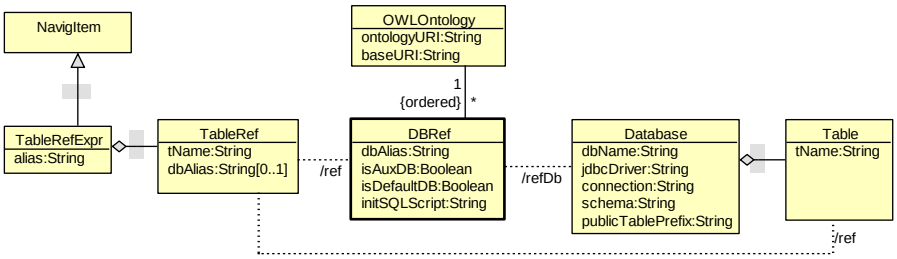


Figure 40. Database and table reference metamodel

For example, suppose we have two databases pointed to from *DBRef* by attribute values *dbAlias='A'* and *dbAlias='B'* respectively and that we have a table expression:

```
A.STUDENT s, B.TEACHER t; s.idcode=t.idcode
```

that uses database aliases *A* and *B* to refer to two tables from different databases and have regular expression aliases *s* and *t*:

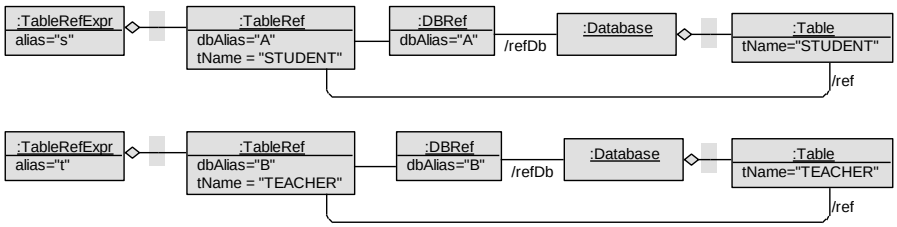
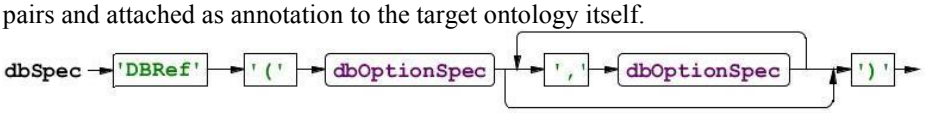


Figure 41. Database and table reference metamodel instance

Syntactically database information is described as a list of attribute name and value pairs and attached as annotation to the target ontology itself.



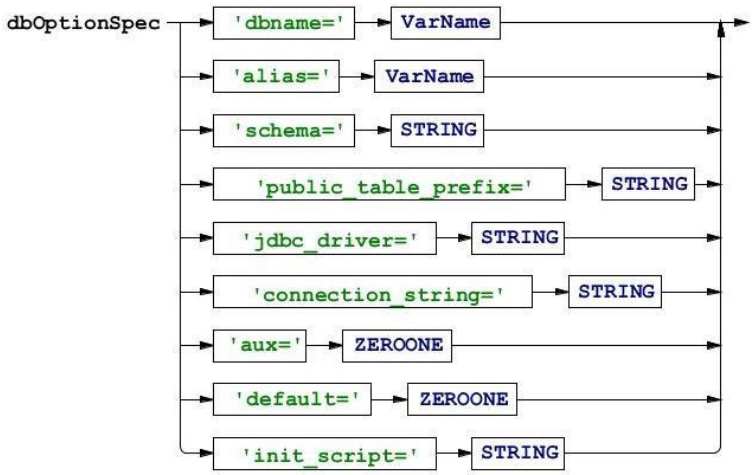


Figure 42. Database description syntax

Example of database description:

```
DBRef(
  alias='M', dbname='school',
  jdbc_driver='com.microsoft.sqlserver.jdbc.SQLServerDriver',
  connection_string='jdbc:sqlserver://GUNTARS-
PC:1433;databaseName=school;user=school;password=s',
  schema='dbo', aux=0, default=1,
  init_script='c:\rdb2owl\sql\RDB2OWL_Init.sql',
  public_table_prefix='school'
)
```

4.3.3 RDB2OWL functions in general

Possibility of function definition and use increases substantially the abstraction level of programming notation. In practical RDB2OWL mapping use cases the functions have been important e.g. to cope concisely with legacy design patterns present in the source database. A basic RDB2OWL function metamodel is shown in **Figure 43**.

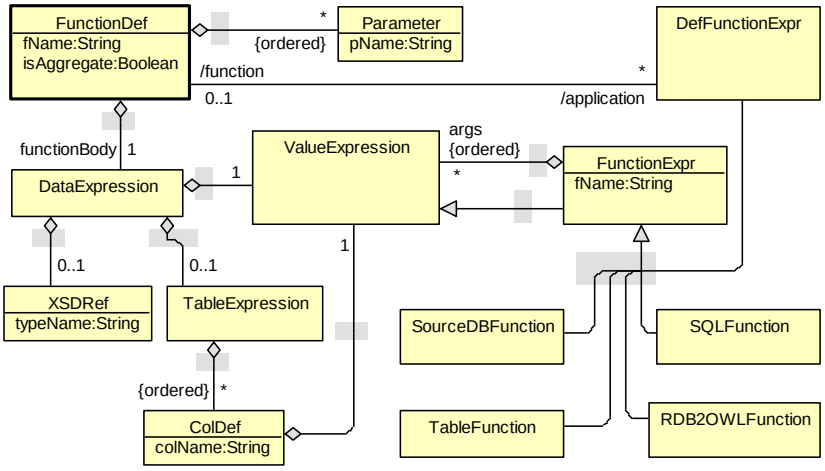


Figure 43. RDB2OWL Function metamodel

We introduce scalar-argument as well as aggregate functions into RDB2OWL (aggregate functions are shown in **Figure 46** and are described in section 4.3.6). The scalar-argument (non-aggregate) functions in RDB2OWL are:

1) built-in functions (class *RDB2OWLFunction*),

2) user defined functions (class *FunctionDef* for definition and associated class *DefFunctionExpr* for application),

3) functions based on stored functions in the source database (class *SourceDBFunction*),

4) functions whose argument-value pairs are stored in table with two columns (class *TableFunction*) and

5) SQL functions (class *SQLFunction*).

4.3.4 Built-in functions

Some functions are frequently needed in various mapping cases. For example, SQL numeric literals 1 / 0 generally are used for Boolean *true* / *false* values therefore we have reason to build-in the *iif* function. For every mapping case the ultimate target is generated triples set, the function *uri* may be helpful for custom URI pattern definition. Built-in function names are prefixed by # to distinguish from user-defined functions. Function parameter names are prefixed by @.

Table 16. RDB2OWL built-in functions

#varchar(@a)	Converts a single argument to SQL varchar type
#xvarchar(@a)	Converts a single argument to varchar, eliminates leading and trailing spaces
#concat(...)	Takes any number of arguments, converts them into the SQL varchar type and then concatenates
#xconcat(...)	Takes any number of arguments, converts into the SQL varchar

	type, eliminates leading and trailing spaces and then concatenates
#uri(@a)	Converts a single argument to varchar, eliminates leading and trailing spaces, converts to Uri encoding
#uriConcat(...)	Takes any number of arguments, converts them into the SQL varchar type, eliminates leading and trailing spaces, converts to Uri encoding and then concatenates
#exists(...)	Can take any number of arguments and returns 1, if at least one argument is not null, otherwise returns 0. The form *#exists(Col1, Col2,..,Colk)* is used in Latvian Medicine registry case as an alternative to multiclass conceptualization approach, if k is small.
#iif(@a,@b,@c)	Chooses the value of @b or @c depending on @a value being 1 or 0. Example: *#iif(is_resident,'true','false')*
#all(...)	Can take any number of arguments and returns 1, if all arguments are not null, otherwise returns 0.

4.3.5 User defined functions

An important feature of RDB2OWL is possibility for user-defined functions, which can be, referenced from class map and property map definitions. Function value is obtained by evaluating its value expression in the context of the function call. The definition of a simple function (e.g., *f(@x)=2*@x+1*) consists just of value expression, referring to function parameters. For simple functions, no *TableExpression* instance is linked to *DataExpression* instance (see metamodel in **Figure 43**).

A user-defined function, however, may include also a table expression (an additional data context for expression evaluation) and a list of column expressions (=calculated columns) relying both on function's arguments and function's table expression and used in further value expression evaluation.

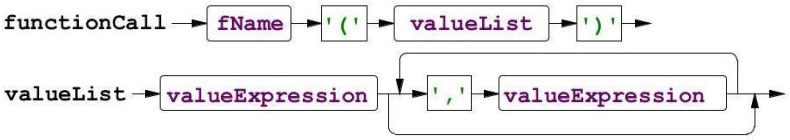


Figure 44. Function call syntax

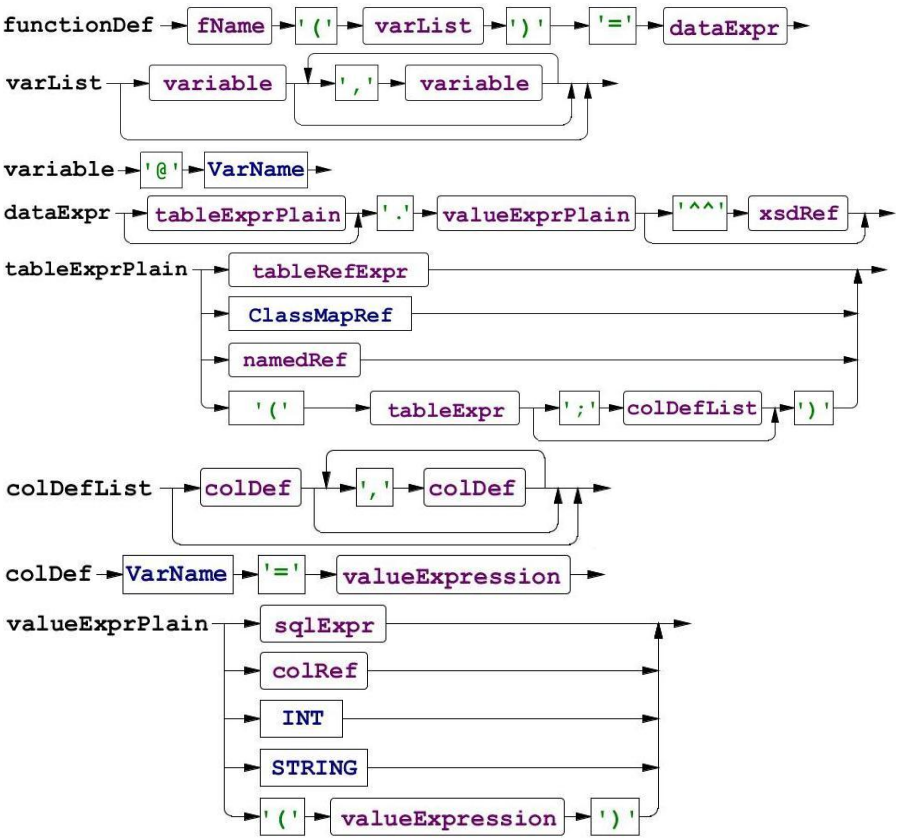


Figure 45. Function definition syntax

Syntactically we have the function definition in the following form:

f(@X₁,@X₂, ...,@Xₙ)=(T; filter; colDef₁, ...,colDefₘ).val ^^xsd_datatype

where *T;filter* is a table expression and each $colDef_i$ is in form $var_i=e_i$ for a value expression e_i. The table expression with column definitions, as well as optional datatype specification (*^^xsd_datayppe*) may be omitted. When the defined function is called as $f(V_1, V_2, ..., V_n)$ in some table context *A*, it is evaluated as:

(A,T;filter'; colDef'₁, ...,colDef'ₘ).val[V₁/@X₁, ..., Vₙ/@Xₙ], where $[V_1/@X_1, ..., V_n/@X_n]$ means substitution of the value V_i for the variable $@X_i$ for all i,

filter'= filter[V₁/@X₁, ..., Vₙ/@Xₙ] and each $colDef'_i = colDef_i [V_1/@X_1, ..., V_n/@X_n]$.

As simple function example with no tables attached is function that converts integer values of 0 / 1 to *'true^^xsd:Boolean' / 'false ^^xsd:Boolean'* is:

BoolT(@X) = #iif (@X,'true','false')^^xsd:Boolean.

Another simple example is: *Plus(@X, @Y) = @X + @Y.*

In Latvian Medical registries there have been numerous situations where many year values were stored in one varchar type field value (e.g., '199920012005') but

corresponding data property having separate instances for each value {‘1999’, ‘2001’, ‘2005’}. The value splitting can be implemented by joining the source table with auxiliary table *Numbers* having single integer type column filled with values from 1 to 999 (see [76]), as in the function:
*split4(@X)=((Numbers;len(@X)>=N*4).substring(@X,N*4-3,4)),* The application *split4(FieldX)* then splits character string into set of substrings of length 4.

If calculated values $colDef_i$: $var_i=e_i$ are included in the function definition, these can be referenced from the function’s value expression. This enables to write more structured and readable code. A simple example function that takes values from two tables and stores intermediate values in variables *courseName* and *teacherName* is:

```
FullCourseInfo(@cId)=((XCourse c)->(XTeacher t); c.AutoId=@cId;
courseName=#concat(c.CName, #iif(c.required,’ required’,’ free’)),
teacherName=t.TName)
.#concat(courseName, ’ by ’, teacherName)
```

We can look on the database table with two columns $T(C_1, C_2)$ as a storage structure with rows containing argument-value pairs of some function *f*. We call this function *f* a *table function* based on table *T*. If the column C_1 has a unique constraint (e.g., a primary key column), *f* is a single-valued function; otherwise, *f* is multi-valued function. Multi-valued functions are appropriate for property maps of properties with cardinality larger than 1.

A table function based on table $T(C_1, C_2)$ actually is shorthand of user-defined function with table expression comprising table *T*: $f(@X) = (T; C_1=@X).C_2$

A typical usage of table functions is for classifier tables containing code and value columns. For example, to associate country codes with full country names a table *Country(code varchar(2), description varchar(40))* with data {(‘de’,’Germany’), (‘en’,’England’), (‘lv’,’Latvia’),…} could be used for a table function.

4.3.6 Aggregate functions

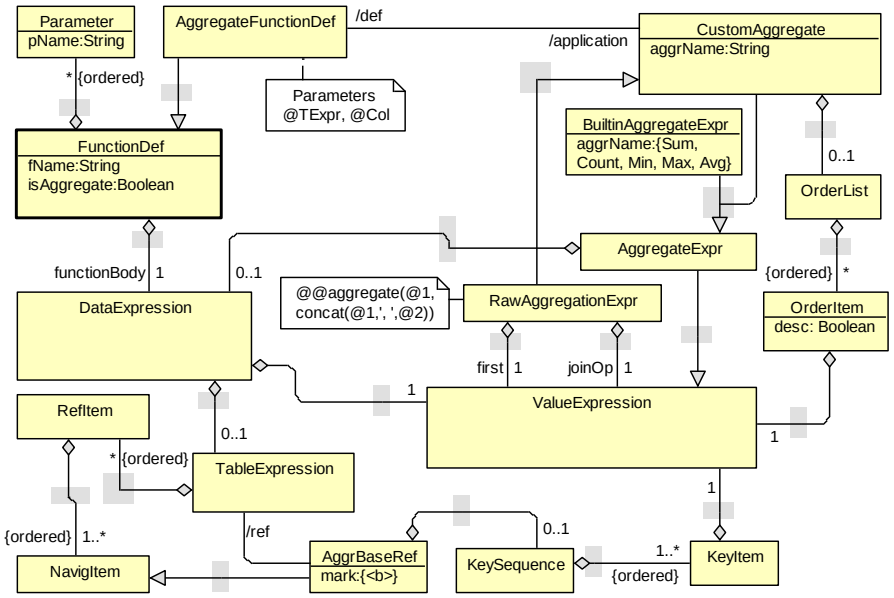


Figure 46. RDB2OWL Aggregate Function metamodel

RDB2OWL has aggregate functions (built-in and user defined) whose application to appropriate arguments yield aggregate expressions (*AggregateExpr* instances). Aggregate expression is kind of value expression therefore it can be substituted for value part of data property: *T.<aggregate expression>*, where table expression *T* is context in which aggregate function is calculated; we call it a *base table expression* of aggregate function application. An aggregate expression specifies 2 things: which aggregate function *f* to execute (built in S*um, Count, Min, Max, Avg* or user defined) and what data should be passed to *f* in terms of data expression *D* (*DataExpression* instance) that contain an optional table expression *E* (*TableExpression* instance) and a value expression *V* (*ValueExpression* instance).

With these denotations an aggregate expression application takes a form: *T.f(E.V)*, where base table expression *T* is explicitly or implicitly referenced by *<b>*-mark from within *E* table expression reference structure. A base table expression *T* can be thought of as a starting point from which table reference or navigation list of E are started to get to the table in which the value expression *V* is evaluated as an argument for the aggregate function *f*. For example, to calculate total salary for a person where Person-to-Work tables are in 1:n relation one would write data property map expression in one of the forms:

```
Person.Sum(<b>=>Work.Salary)
Person.Sum(=>Work.Salary)
Person.Sum(
(<b> {key=(PersonID)}, Work w; <b>.PersonID=w.PersonID).Salary )
```

In this example *Person* is the base table referenced by <b> (omitted in the short form). The longest form shows the use of explicit key sequence (*KeySequence, KeyItem* instance) that specifies grouping by option. When key sequence is omitted, the primary key column sequence for base table expression is assumed. The above example expression can translate into an execution environment as:

```
SELECT sum(Salary)
FROM Person p, Work w
WHERE p.PersonID=w.PersonID
GROUP BY p.PersonID
```

In the mini-University example, to calculate the course count that each teacher teaches, one can use *[[Teacher]]* notation to refer to the sole class map for the *Teacher* class, thus writing:

[[Teacher]].Count(<b> {key=(teacher_id)} => Course.course_id)

RDB2OWL has built-in function *@@aggregate* (*RawAggregationExpr* instance) offering custom aggregate expression definitions. *@@aggregate* takes 4 arguments:

- a table expression, including a reference to the <b>-tagged context expression and a defined key list (within the <b>-tagged expression), and optionally an order by clause;
- a value expression to be aggregated over
- a single argument function for first value processing in the aggregate formation (the sole variable for this function is denoted by @1)
- a two argument function for adding the next value to the aggregate (the value accumulated so far is denoted by @1 and the next value is denoted by @2).

For example, to get the course list (comma-separated code list) each student is registered to, one would write:

```
[[Student]].@@aggregate((<b>=>Registration-> Course {Code asc}),
Code, @1, #concat(@1, ', ', @2) ).
```

User defined aggregate functions (*AggregateFunctionDef* instance) can be defined with *@@aggregate* function. If variables named *@TExpr* (denotes a table expression) and *@Col* (denotes columns for value expression) are present in the context of the call to *@@aggregate*, the first two arguments in the call may be omitted, they are filled by the values of these variables. This allows shorter forms of user-defined aggregate function definition ('@@' is the name prefix for user-defined aggregate functions):

```
@@List( @TExpr,@Col ) =  @@aggregate( @1, #concat(@1, ', ', @2) )
```

Shorter forms of aggregate function definition omit variables *@TExpr* and *@Col*:

```
@@List()= @@aggregate( @1, #concat(@1, ', ', @2) ).
```

To get course list one can apply this user-defined aggregate function @@List:

```
[[Student]].@@List((=>Registration->Course {Code asc}).Code)
```

In this example the table expression

```
=>Registration->Course {Code asc}
```

is assigned to variable *@TExpr* and the value expression *Code* – to variable *@Col*.

4.3.7 Extended mapping example

We present an example illustrating the advanced RDB2OWL construct application. **Figure 47** and **Figure 48** show extended mini-University DB schema and target ontology example with mapping annotations. Ontology level annotations describe two database schemas- one for source database (referenced by 'M') and auxiliary database

(referenced by 'A') for which SQL script *RDB2OWL_init* is specified to be executed before starting of mapping processing for triple generation. The list of user-defined function definitions is located also in ontology level annotations. Note that definition for *split* function references auxiliary database A where auxiliary table Numbers resides. This function *split* splits a coma separated value into its parts, e.g., '11,12,13'→{'11','12','13'}, its definition uses another RDB2OWL function *encoma* that puts comas around string value ('11'→',11,').

Aggregate built-in function *Sum* is applied to define data property map for property *creditsTaken*. Because expression *Sum(=>XRegistration->XCourse.Credits)* omits an explicit base table expression it is assumed the one defined for the sole class map of *Student* class, which is *XStudent*. Expanded form would be *XStudent.Sum(<b>{key=(AutoID)}=>XRegistration->XCourse.Credits)*. Aggregate expression for *creditsPaid* property is defined similarly; it uses also row-filtering condition.

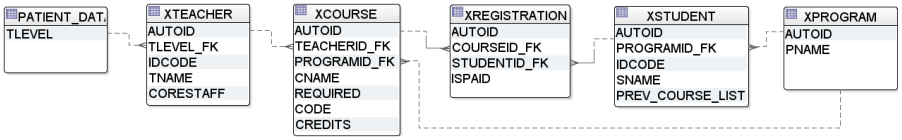


Figure 47. An extended mini-university database schema

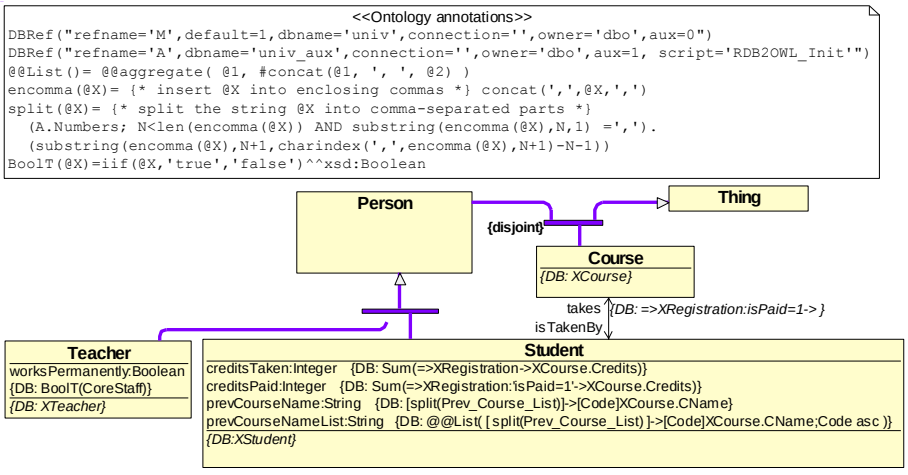


Figure 48. Extended ontology example

In this extended mini-University example table *XStudent* has column *prev_course_list* to hold comma-separated list of codes of previous course list, e.g., '*semweb,prog01,prog02,softeng*' (not a good database design but ontologies should be map-able to real databases). In ontology *prevCourseName* property is with cardinality larger than 0. The mapping expression for this property *[split(Prev_Course_List)]->[Code]XCourse.CName* specifies 2-step transformation:

1) *split(Prev_Course_List)* means splitting of comma-separated value in *prev_course_list* into separate parts: ‘semweb,prog01,prog02,softeng’→{‘semweb’, ’prog01’, ’prog02’, ‘softeng’},

2) separate value list from step (1) is put into navigation link structure as column value to join with *XCourse* table to get name list from code list, e.g., *[{‘semweb’, ’prog01’, ’prog02’, ‘softeng’}]->[Code]XCourse.CName→{‘Semantic Web’, ‘Programming 1’, ‘Programming2’, ‘Software Engineering’}*.

Data property map expression for property *prevCourseNameList* is a bit more complicated: the same expression as for property *prevCourseName* is put as argument in application of user defined *@@List* function to obtain the list of previous course names into comma-separated string.

5 RDB2OWL mapping implementation using relational schema

RDB2OWL Raw and Core languages (described in Section 4) allow for description of RDB-to-OWL/RDF mappings but there is a need for possibility to execute such defined mappings to obtain RDF triples corresponding to the target OWL ontology. Therefore we introduce special execution framework whose main idea is storing mapping data in relational database schema (we call it: mapping RDB schema) and capability to use the power of modern relational database engines in SQL execution.

More specifically, we introduce a simple standard SQL-based RDB to RDF/OWL mapping approach that is based on defining correspondence between the tables of the database and the classes of the ontology, as well as, between table fields/links in the database and data/object properties in the ontology with possible addition of filters and linked tables in the mapping definition. This mapping formalism allows an automatic generation of SQL statements that generate the RDF triples that correspond to the source database data.

The mapping RDB schema is designed to have similarities to RDB2OWL Raw and Core languages. The purpose for this is to make it possible to transform the user-defined mappings expressed in RDB2OWL Raw or Core languages into execution environment (data in mapping RDB schema tables) for triple generation.

5.1 The mapping execution framework

Figure 1 shows the architecture of RDB-to-RDF/OWL mapping process in the RDB2OWL framework.

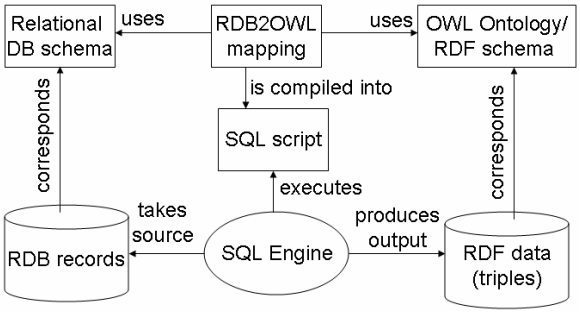


Figure 49. RDB2OWL framework architecture

The task of the RDB2OWL mapping is to establish a correspondence between a relational database schema (or several schemas) and elements (entities) of OWL ontology (its schema part) or RDF schema (a single mapping can involve possibly

several OWL ontologies), so that the corresponding RDB records could be translated (dumped) into RDF triples that correspond to the given ontology or RDF schema.

The process leading to generation of RDF triples for the target ontology consists of two phases. In the first step the RDB2OWL mapping information that is stored in relational database is processed essentially by SQL commands to generate another SQL scripts for RDF triples generation. In the second step, the generated SQL scripts are executed in the source database to get RDF triples that correspond to the target ontology.

5.2 Mapping schema description and its semantics for triple generation

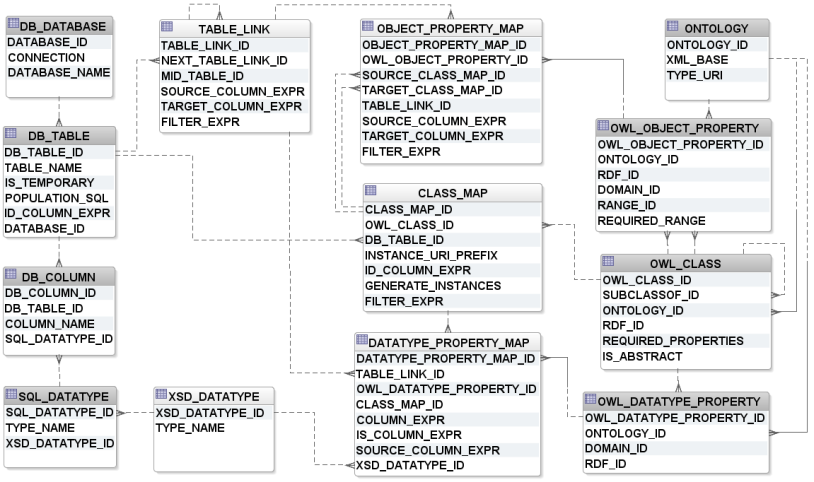


Figure 50. RDB2OWL mapping RDB schema

We assume that the relational database and OWL ontology are given - this is a typical situation where OWL ontology is an end-user-oriented representation of the data contained in RDB. Figure 50 shows relational database schema that stores information about the RDB2OWL mappings. The mapping information relies on the source database schema description stored in the *db_database, db_table* tables and *db_column* and the target OWL ontology or RDF schema description stored in tables *ontology*, *owl_object_property* and *owl_datatype_property*. Only part of database or ontology information is stored in the mapping schema. The URI of ontology entities is obtained by concatenating *ontology.xml_base* field with *rdf_id* field from *owl_class, owl_datatype_property* or *owl_object_property* tables.

The mappings are specified in records of tables *class_map*, *object_property_map, datatype_property_map*. For a *class_map* record *x* we call a *base table* of *x* a source database table that is specified in *db_table* record linked to *x.* Each record *r* in the *class_map* table specifies triple generation of the form *<s,'rdf:type',o>*, where *o* is URI of the *owl_class* record linked to *r*. The URI of the subject *s* in the above triple is

formed using the *instance_uri_prefix* in *r*, concatenated to the value of the expression specified in *r* in *id_column_expr* evaluated in records of the base table *t* of *r*. If *filter_expr* is specified in *r*, then only those records of *t* that satisfy it are considered for the subject's *s* generation. There are possibly several *class_map* records corresponding to a single *owl_class* record. The *class_map* records with *generate_instances=0* are not used for the triple generation but may later be referenced from property maps.

A record *ro* of *object_property_map* table specifies generation of triples <*s,p,o*> where the predicate *p* is the URI of the *owl_object_property* record linked to *ro*. We let *src* and *trg* to denote the *class_map* records that are referred in *ro* via the *source_class_map_id -> class_map_id* and *target_class_map_id -> class_map_id* links, respectively. Let, furthermore, *t_src* and *t_trg* be the base tables of *src* and *trg*, respectively. The *s* and *o* values in the above triple are obtained from all rows in the join of *t_src* and *t_trg* on the equality of columns specified in *ro* fields *source_column_expr* and *target_column_expr*, further filtered by *ro*'s *filter_expr*. Similarly, a *datatype_property_map* record *rd* specifies generation of triples <*s,p,o*>, where *p* is URI from the linked *owl_datatype_property* record. Let *src* be the *class_map* record that is linked to *rd*. Then *s* and *o* are obtained from each row of *src*'s base table – *s* by means of class map URI formation and *o* as a value of *ro*'s *column_expr* expression.

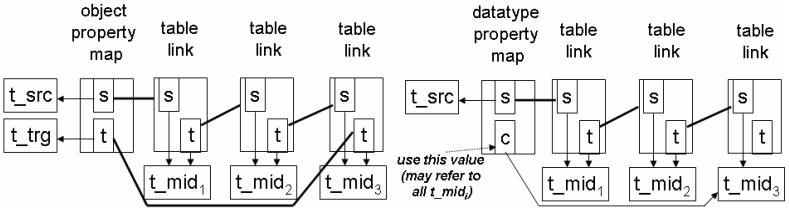


Figure 51. Intermediate tables in property maps' definitions

The *table_link* table allows introducing intermediate steps in table joining in object property definition, as well as auxiliary linked tables in data property definition. Figure 3 sketches the table linking schema in case of *table_link* usage (s stands for *source_column_expr, t* for *target_column_expr* and c for *column_expr*; the arrow stands for column belonging to a table, and the bold line for equality condition; note that each iteration via *table_link* introduces a new table into the join expression).

The table below outlines some class map information (with linked OWL class *rdf_id* and DB table name).

Table 17. Some class map information for Mini-University example

class_map_id	OWL klase (rdf_id)	table_name	filter_expr	id_column_expr	instance_ uri_ prefix	generate_ instances
1	Teacher	teacher		teacher_id	Teacher	0
2	Student	student		student_id	Student	1
3	Course	course		course_id	Course	0
4	Mandatory Course	course	required=1	course_id	Course	1
5	PersonID	teacher		idcode	PersonID	1

6	PersonID	student		idcode	PersonID	1

The tables below outline some data and object property maps (*s* and *t* denote source and target class map id's).

Table 18. Some property map information for Mini-University example

s	**datatype_property.rdf**	**table_name**	**column_expr**	**filter_expr**
3	courseName	course	name	
1	personName	teacher	name	
2	personName	student	name	

s	**t**	**object_property**	**table_name(s)**	**table_name(t)**	**source_col_expr**	**target_col_expr**
2	6	personID	student	student	student_id	student_id
1	5	personID	teacher	teacher	teacher_id	teacher_id
1	3	teaches	Teacher	course	teacher_id	teacher_id

Note that the maps for data and object properties can refer to class maps with id's 1 and 3 that are not used for class instance triple generation.

5.3 Advanced mapping schema features

There are cases when a large database table, say *t*, is modeled by a set of OWL classes, each class *c* responsible for a certain subset of the table columns (e.g. there are different groups of measurements taken during a clinical anamnesis, all recorded into a single table). Mapping such table onto all the classes, would require writing lengthy filtering conditions involving all the group columns.

Our proposal is to introduce into the mapping schema explicit features allowing to specify generation of only those <*x*,'rdf:type',*o*> instance triples, where a generated triple <*x*,*p*,*y*> exists for some property *p* whose domain is *o*. This allows us to keep the simple mapping from *t* to all classes *c* associated to parts of *t* with no filtering. We specify this requirement by *owl_class.required_properties=1* and implement by deleting triples without property instances in the 2nd phase of the mapping generation. Actually, this is implementation of multiclass conceptualization of RDF2OWL Core Plus mapping language described in section 4.3.1. This feature is extensively used in the mapping definition for Latvian medical registries case (for 54 out of 172 OWL classes; 542 out of 814 OWL data properties on them).

The field *required_range* in *owl_object_property* table specifies requirement to delete those OWL object property instance triples <*x,op,y*> for which there are no <*y*,'rdf:type',*r*> triple with *r* being the range of *op*, after the *required_properties* optimization.

Auxiliary tables and temporary tables can be introduced allowing to use standard SQL for mapping specification. We have used a few auxiliary tables – the tables for classifiers not properly introduced in source database schema; and the Numbers table (with all numbers from 1 up to 999) used for field value splitting into multiple data property values (we used this pattern for 111 datatype property maps). Auxiliary tables can be placed in a different database pointed to from *database* table row.

5.4 RDF triple generation

5.4.1 Class instance triple generation

We show triple generation on Mini-university example (described in Section 2.3.1). In the **Table 19** below there are listed OWL class mappings. The example uses only mappings to database tables, listing data from the *class_map* tables and referenced tables *owl_class, db_table.*

Table 19. OWL class maps to database tables

class_ map_ id	OWL class (rdf_id)	table_name	filter_expr	id_column_ expr	instance_ uri_ prefix	generate_ instances
1	Teacher	teacher		teacher_id	Teacher	0
2	Assistant	teacher	level_code= 'Assistant'	teacher_id	Teacher	1
3	AssocProfessor	teacher	level_code= 'Associate Professor'	teacher_id	Teacher	1
4	Professor	teacher	level_code= 'Professor'	teacher_id	Teacher	1
5	Student	student		student_id	Student	1
6	Course	course		course_id	Course	0
7	MandatoryCourse	course	required=1	course_id	Course	1
8	OptionalCourse	course	required=0	course_id	Course	1
9	PersonID	teacher		idcode	PersonID	1
10	PersonID	student		idcode	PersonID	1
11	AcademicProgram	program		program_id	Program	1

Most of the class mappings are used for the real OWL class instance generation. There are some class mappings not used in class instance generation, but which will be further referenced in data property mappings.

With mere SQL statement, it is possible to generate another SQL statement which executed in *sample DB* would generate instance RDF triples. Executing script *OWL_instance_gen.sql* (see Appendix A.6.1 for code) against our sample data, we obtain row set with generated SQL statements, one of which is:

```
SELECT
  '<lumii#COURSE' + CAST(t.COURSE_ID AS varchar) + '>' as subject,
  '<type>' as predicate,
  '<lumii#MandatoryCourse>' as object
FROM COURSE t
WHERE required=1
```

Executing all generated statements in our sample *source DB,* we obtain the following triples, duplicates removed. The duplicates in the example come from the fact that one *teacher* table row and one *student* table row have the same *idcode* value (the same person being student and teacher at the same time). For brevity and readability considerations, we use in **Table 20** and the following tables as well in

SQL source codes the prefix “lumii” to denote “http://lumii.lv/ex”, and the predicate notation “type” to stand for http://www.w3.org/1999/02/22-rdf-syntax-ns#type.

Table 20. Generated OWL class instance RDF tripples

Subject	Predicate	Object
< lumii #Course1>	<type>	<lumii#OptionalCourse>
< lumii #Course2>	<type>	<lumii#MandatoryCourse>
< lumii #Course3>	<type>	<lumii#MandatoryCourse>
<lumii#Course4>	<type>	<lumii#OptionalCourse>
<lumii#PersonID123456789>	<type>	<lumii#PersonID>
<lumii#PersonID345453432>	<type>	<lumii#PersonID>
<lumii#PersonID555555555>	<type>	<lumii#PersonID>
<lumii#PersonID777777777>	<type>	<lumii#PersonID>
<lumii#PersonID987654321>	<type>	<lumii#PersonID>
<lumii#PersonID999999999>	<type>	<lumii#PersonID>
<lumii#Program1>	<type>	<lumii#AcademicProgram>
<lumii#Program2>	<type>	<lumii#AcademicProgram>
<lumii#Student1>	<type>	<lumii#Student>
<lumii#Student2>	<type>	<lumii#Student>
<lumii#Student3>	<type>	<lumii#Student>
<lumii#Student4>	<type>	<lumii#Student>
<lumii#Teacher1>	<type>	<lumii#Professor>
<lumii#Teacher2>	<type>	<lumii#Professor>
<lumii#Teacher3>	<type>	<lumii#Assistant>

5.4.2 OWL datatype property value triple generation

Table 21 below shows data in the *datatype_property_map* table and referenced tables *class_map* and *owl_datatype_property* in case when no table link is used (no *table_link* table usage). One can compare the first column in Table 7 and Table 9 below. For example, property *personName* is linked to *class_map_id*=1 and *class_map_id*=5 that correspond to class maps for OWL classes *Teacher* and *Student*. For *Teacher* class instances are not directly generated (*generate_instances=0*). Class instances are generated for the *Professor, AsocProfessor* and *Asistant* subclasses. In addition, as far as *instance_uri_prefix, table_name* and *id_column_expr* have the same value in the class map for superclass (*Teacher* in this case), this enable correct generation of the subject part of triples for OWL data properties. There is no need to make class map for each subclass. As to correctness of the mapping, the class map to super class should have the same filtering as union of all subclasses. In the case of *Teacher* it has no filter (*filter_expr* is empty for *class_map_id*=1) but filters for sub-class maps (*class_map_id:2,3,4*) are *level_code='Assistant'*, *level_code='Associate Professor*' and *level_code='Professor'*. All these together give all *teacher* rows and *Teacher* class map with no filtering correspond to the same row set.

Table 21. OWL datatype property class mappings to database table column expressions (data from tables *datatype_property_map* and referenced *class_map, db_table* and *owl_datatype_property*)

class_map_id	OWL_datatype_property	table_name	column_expr	filter_expr
6	courseName	Course	name	
11	programName	Program	name	
1	personName	Teacher	name	
5	personName	Student	name	
9	IDValue	Teacher	idcode	
10	IDValue	Student	idcode	

Executing script *generate_sql4datatype_props.sql* (see Appendix [A.6.2] for code) against our sample data, we obtain row set with generated SQL statements one of which is:

```
SELECT
  '<lumii#COURSE' + CAST(t.COURSE_ID AS VARCHAR) + '>' as subject,
  '<lumii#courseName>' as predicate,
  '"'+  CAST(name AS varchar)+ '"^^xsd:string' as object
FROM COURSE t
WHERE name IS NOT NULL
```

Executing all generated statements in our sample *source DB* we obtain the following triples, duplicates and "xsd:string" from Object part removed:

Table 22. Generated OWL datatype property instance RDF triples

Subject	Predicate	Object
<lumii#Course1>	<lumii#courseName>	Programming Basics
<lumii#Course2>	<lumii#courseName>	Semantic Web
<lumii#Course3>	<lumii#courseName>	Computer Networks
<lumii#Course4>	<lumii#courseName>	Quantum Computations
<lumii#PersonID123456789>	<lumii#IDValue>	123456789
<lumii#PersonID345453432>	<lumii#IDValue>	345453432
<lumii#PersonID555555555>	<lumii#IDValue>	555555555
<lumii#PersonID777777777>	<lumii#IDValue>	777777777
<lumii#PersonID987654321>	<lumii#IDValue>	987654321
<lumii#PersonID999999999>	<lumii#IDValue>	999999999
<lumii#Student1>	<lumii#personName>	Dave
<lumii#Student2>	<lumii#personName>	Eve
<lumii#Student3>	<lumii#personName>	Charlie
<lumii#Student4>	<lumii#personName>	Ivan
<lumii#Teacher1>	<lumii#personName>	Alice
<lumii#Teacher2>	<lumii#personName>	Bob
<lumii#Teacher3>	<lumii#personName>	Charlie
<lumii#Program1>	<lumii#programName>	Computer Science
<lumii#Program2>	<lumii#programName>	Computer Engeneering

5.4.3 OWL object property value triple generation

In **Table 23** data from *object_property_map,* and referenced *owl_object_property*, as well as two *class_map* rows for subject and object and corresponding *db_table* rows are shown. See **Table 19** for details on referenced *class_map* rows.

Table 23. owl object property mappings to database tables pairs for domain and range

class_ map_id (domain)	class_ map_id (range)	object_ property	table_name (domain)	table_name (range)	source_ col_expr	target_ col_expr
11	6	includes	program	course	program_id	program_id
5	10	personID	student	student	student_id	student_id
1	9	personID	teacher	teacher	teacher_id	teacher_id
5	11	enrolled	student	program	program_id	program_id
1	6	teaches	teacher	course	teacher_id	teacher_id

As data shows OWL object properties generally map to table pairs corresponding to domain and range class pair. Exception is *personID* object property because it has *Person* class as domain and *PersonID* class as range and both these classes has mapping to 2 tables: *student* and *teacher.* For this property there exist two mappings (*object_property_map* rows) one of which maps *student* table for domain to *student* table for range and the mapping is based on *student_id* column (*source_column_expr*, *target_column_expr*). The second row maps *teacher* table to *teacher* table based on *teacher_id* column in a similar way.

To generate RDF triples for OWL object property instances the data shown above in Table 21 can be used. A skeleton of SQL for main information retrieval for generation process is, as follows:

```
SELECT
  t_domain.<domain_class_map_id→class_map.id_column_expr>,
  t_range.<range_class_map_id→class_map.id_column_expr>
FROM <domain_table> AS t_domain
  INNER JOIN <range_table> t_range
    ON t_domain.<domain_column_expr>
       = t_range.<range_column_expr>
```

The aliases *t_domain* and *t_range* are used to prevent name collision if table name for domain and range class maps are the same. For example, in the case of one mapping for *PersonID* property (for *student*) query joins *student* table to itself because *object_property_map* table specifies two tables via *domain_class_map* and *range_class_map* although the tables are the same. Data for *PersonID* Object property:

```
SELECT t_domain.student_id, t_range.idcode
FROM student AS t_domain
INNER JOIN student AS t_range
  ON t_domain.student_id = t_range.student_id
```

For *enrolled* property the query essentially is

```
SELECT t_domain.student_id, t_range.program_id
FROM student AS t_domain
  INNER JOIN program AS t_range
  ON t_domain.program_id = t_range.program_id
```

An SQL script that generates instances for OWL object properties can be defined in a similar way, as for OWL class and OWL data property instance generation.

Executing script *generate_sql4object_props.sql* (see Appendix A.6.3 for code) against our sample data, we got row set with generated SQL statements one of which was:

```
  SELECT
  '<lumii#PROGRAM'
    + CAST(t_domain.PROGRAM_ID AS varchar) + '>' as subject,
  '<lumii#includes>' as predicate,
  '<lumii#COURSE' + CAST(t_range.COURSE_ID AS varchar) + '>' as object
FROM
  PROGRAM t_domain
  INNER JOIN COURSE  t_range
     ON  t_domain.PROGRAM_ID = t_range.PROGRAM_ID
WHERE 1=1 AND 1=1
```

Executing all generated statements in our sample *source DB* we got following triples:

Table 24. Generated OWL object property instance RDF triples

Subject	Predicate	Object
<lumii#Student1>	<lumii#enrolled>	<lumii#Program1>
<lumii#Student2>	<lumii#enrolled>	<lumii#Program2>
<lumii#Student3>	<lumii#enrolled>	<lumii#Program1>
<lumii#Student4>	<lumii#enrolled>	<lumii#Program2>
<lumii#Program1>	<lumii#includes>	<lumii#Course4>
<lumii#Program1>	<lumii#includes>	<lumii#Course2>
<lumii#Program2>	<lumii#includes>	<lumii#Course1>
<lumii#Program2>	<lumii#includes>	<lumii#Course3>
<lumii#Student1>	<lumii#personID>	<lumii#PersonID123456789>
<lumii#Student2>	<lumii#personID>	<lumii#PersonID987654321>
<lumii#Student3>	<lumii#personID>	<lumii#PersonID555555555>
<lumii#Student4>	<lumii#personID>	<lumii#PersonID345453432>
<lumii#Teacher1>	<lumii#personID>	<lumii#PersonID999999999>
<lumii#Teacher2>	<lumii#personID>	<lumii#PersonID777777777>
<lumii#Teacher3>	<lumii#personID>	<lumii#PersonID555555555>
<lumii#Teacher1>	<lumii#teaches>	<lumii#Course2>
<lumii#Teacher2>	<lumii#teaches>	<lumii#Course3>
<lumii#Teacher2>	<lumii#teaches>	<lumii#Course4>
<lumii#Teacher3>	<lumii#teaches>	<lumii#Course1>

Now we discuss the table link usage. It is needed for instance generation of OWL object property *takes* which is between *Student* and *Course* OWL classes and need to join the *student* and *course* tables through *registration* table. Table *object_property_map* links to *class_map* two rows for subject and object through *domain_class_map_id* and *range_class_map_id* foreign keys. That gives pair of two relations (tables). To join these tables *source_column_expr* and *target_column_expr* are used. If these tables cannot be joined directly then the *table_link* table can be used. It stores information about middle steps in table traversing. To support joining table *t1* with *t2* through middle table the *table_link* columns has these meanings:

mid_table_name- table name in the middle step,
source_column_expr- joins <*mid_table_name*> table to *t1* by this column expr.,
target_column_expr- joins <*mid_table_name*> table to *t2* by this column expr.,
filter_expr- additional filter expression on table <*mid_table_name*>

next_table_link_id- foreign key to the same table to implement more intermediate steps if needed (t1→mid_table_1→mid_table_2 → … →mid_table_n→t2).

Table 25 and Table 26 show OWL object property mapping data for the *takes* property that needs table links (*object_property_map.table_link* is not null). Data are from tables *owl_object_property, object_property_map* as well as their referenced table rows. The corresponding *table_link* data is shown also. *Filter_expr* is not used in the example.

Table 25. owl object property mappings to database tables pairs for domain and range when table link is used

class_ map_id (domain)	class_ map_id (range)	object_ property	table_name (domain)	table_name (range)	source_ column_ expr	target_ column_ expr
5	6	takes	student	Course	student_id	course_id

Table 26. *table_link* table data

mid_table_name	source_column_expr	target_column_expr	next_table_link_id
registration	student_id	course_id	

The join condition is:

```
<domain_table>.<source_column_expr>=
<mid_table_name>.<table_link.source_column_expr>
AND
<mid_table_name>.<table_link.target_column_expr>=
<range_table>.<target_column_expr>
```

In this case the concrete condition is:

```
student.student_id=registration.student_id
AND
registration.course_id=course.course_id
```

Executing script *generate_sql4object_props_table_links.sql* (see Appendix [A.6.4] for code) against our sample data, we obtain row set with generated SQL statements, consisting of:

```
SELECT
  '<lumii#STUDENT'
    + CAST(t_domain.STUDENT_ID AS varchar) + '>' as subject,
  '<lumii#takes>' as predicate,
  '<lumii#COURSE'
    + CAST(t_range.COURSE_ID AS varchar) + '>' as object
FROM
  STUDENT t_domain
  INNER JOIN REGISTRATION mid1
    ON t_domain.STUDENT_ID = mid1.STUDENT_ID
  INNER JOIN COURSE  t_range
    ON  mid1.COURSE_ID = t_range.COURSE_ID
WHERE 1=1 AND 1=1
```

Executing it in sample *source DB* we get the following triples:

Table 27. Generated OWL object property instance RDF triples when *table_link* table used

Subject	Predicate	Object
<lumii#Student1>	<lumii#takes>	<lumii#Course2>
<lumii#Student2>	<lumii#takes>	<lumii#Course4>

Subject	Predicate	Object
<lumii#Student3>	<lumii#takes>	<lumii#Course1>
<lumii#Student4>	<lumii#takes>	<lumii#Course3>
<lumii#Student5>	<lumii#takes>	<lumii#Course2>

5.4.4 The result of RDF triple generation for Mini-university example

When all generated SQLs were executed in the sample Mini-university database, we get following triple set. Essentially, it is data export from original relational database to RDF format for target OWL ontology. It is union of data showed in Table 20, Table 22, Table 24 and Table 27 with shorthands "lumii" and "type" expanded.

<http://lumii.lv/ex#Course1> <http://lumii.lv/ex#courseName> Programming Basics
<http://lumii.lv/ex#Course2> <http://lumii.lv/ex#courseName> Semantic Web
<http://lumii.lv/ex#Course3> <http://lumii.lv/ex#courseName> Computer Networks
<http://lumii.lv/ex#Course4> <http://lumii.lv/ex#courseName> Quantum Computations
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program1>
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program2>
<http://lumii.lv/ex#Student3> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program1>
<http://lumii.lv/ex#Student4> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program2>
<http://lumii.lv/ex#PersonID123456789><http://lumii.lv/ex#IDValue> 123456789
<http://lumii.lv/ex#PersonID345453432><http://lumii.lv/ex#IDValue> 345453432
<http://lumii.lv/ex#PersonID555555555><http://lumii.lv/ex#IDValue> 555555555
<http://lumii.lv/ex#PersonID777777777><http://lumii.lv/ex#IDValue> 777777777
<http://lumii.lv/ex#PersonID987654321><http://lumii.lv/ex#IDValue> 987654321
<http://lumii.lv/ex#PersonID999999999><http://lumii.lv/ex#IDValue> 999999999
<http://lumii.lv/ex#Program1> <http://lumii.lv/ex#includes><http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Program1> <http://lumii.lv/ex#includes><http://lumii.lv/ex#Course4>
<http://lumii.lv/ex#Program2> <http://lumii.lv/ex#includes><http://lumii.lv/ex#Course3>
<http://lumii.lv/ex#Program2> <http://lumii.lv/ex#includes><http://lumii.lv/ex#Course1>
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#personID> <http://lumii.lv/ex#PersonID123456789>
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#personID> <http://lumii.lv/ex#PersonID987654321>
<http://lumii.lv/ex#Student3> <http://lumii.lv/ex#personID> <http://lumii.lv/ex#PersonID555555555>
<http://lumii.lv/ex#Student4> <http://lumii.lv/ex#personID> <http://lumii.lv/ex#PersonID345453432>
<http://lumii.lv/ex#Teacher1><http://lumii.lv/ex#personID> <http://lumii.lv/ex#PersonID999999999>
<http://lumii.lv/ex#Teacher2><http://lumii.lv/ex#personID> <http://lumii.lv/ex#PersonID777777777>
<http://lumii.lv/ex#Teacher3><http://lumii.lv/ex#personID> <http://lumii.lv/ex#PersonID555555555>
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#personName> Dave
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#personName> Eve
<http://lumii.lv/ex#Student3> <http://lumii.lv/ex#personName> Charlie
<http://lumii.lv/ex#Student4> <http://lumii.lv/ex#personName> Ivan
<http://lumii.lv/ex#Teacher1><http://lumii.lv/ex#personName> Alice
<http://lumii.lv/ex#Teacher2><http://lumii.lv/ex#personName> Bob
<http://lumii.lv/ex#Teacher3><http://lumii.lv/ex#personName> Charlie
<http://lumii.lv/ex#Program1> <http://lumii.lv/ex#programName> Computer Science
<http://lumii.lv/ex#Program2> <http://lumii.lv/ex#programName> Computer Engeneering
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#takes> <http://lumii.lv/ex#Course4>
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#takes> <http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#takes> <http://lumii.lv/ex#Course3>
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#takes> <http://lumii.lv/ex#Course1>
<http://lumii.lv/ex#Student3> <http://lumii.lv/ex#takes> <http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Teacher1><http://lumii.lv/ex#teaches> <http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Teacher2><http://lumii.lv/ex#teaches> <http://lumii.lv/ex#Course3>
<http://lumii.lv/ex#Teacher2><http://lumii.lv/ex#teaches> <http://lumii.lv/ex#Course4>
<http://lumii.lv/ex#Teacher3><http://lumii.lv/ex#teaches> <http://lumii.lv/ex#Course1>
<http://lumii.lv/ex#Course1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#OptionalCourse>
<http://lumii.lv/ex#Course2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#MandatoryCourse>
<http://lumii.lv/ex#Course3> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#MandatoryCourse>
<http://lumii.lv/ex#Course4> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#OptionalCourse>
<http://lumii.lv/ex#PersonID123456789><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID345453432><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#PersonID>

<http://lumii.lv/ex#PersonID555555555><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID777777777><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID987654321><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID999999999><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#Program1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#AcademicProgram>
<http://lumii.lv/ex#Program2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#AcademicProgram>
<http://lumii.lv/ex#Student1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Student2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Student3> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Student4> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Teacher1><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Professor>
<http://lumii.lv/ex#Teacher2><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Professor>
<http://lumii.lv/ex#Teacher3><http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://lumii.lv/ex#Assistant>

5.4.5 Use of chain of linked tables

Any number of linked tables can be attached to property map to allowing to join domain table and range tables (attached to domain and range class maps) through intermediate tables for object property map. For data property map linked tables allows to specify a value expression based on columns in linked tables.

Denote by *opm* a row of *object_property_map* table and *tl(i), i=1,2,...* the rows of *table_link* table linked to *opm:* *opm→tl(1)→ tl(2)→... → tl(n)*. This schema introduces:

source table: *omp.source_class_map_id→class_map→db_table.table_name*

for each *i: mid_table(i)=tl(i).mid_table_id→db_table.table_name*

target table: *omp.target_class_map_id→class_map→db_table.table_name*

The tables are linked in chain:

```
source_table→
mid_table(1)→ mid_table(2)→... → mid_table(n)
→target_table
```

The tables are joined (see **Figure 51**) by conjunction (and) of conditions:

```
opm.source_column_expr=tl(1).source_column_expr
tl1(1).target_column_expr=tl2.source_column_expr
tl1(2).target_column_expr=tl3.source_column_expr
...
tl1(n-1).target_column_expr=tl(n).source_column_expr
tl1(n).target_column_expr=opm.source_column_expr
```

We use naming convention for aliases *t_domain*, *mid1*, *mid2*, ...,t_*range* for *source_table*, *mid_table(1)*, *mid_table(2)*, ..., *target_table* respectively. Users can specify *s* for *t_domain* and *t* for *t_range* for brevity.

If the chain of table link has some link pair that cannot be linked on equality of column expressions but on Boolean expression including columns from both tables then *filter_expr* column is used:

opm.filter_expr contain condition to join *source_table* with *mid_table(1)*
tl(1).filter_expr contain condition to join *mid_table(1)* with *mid_table(2)*
tl(2).filter_expr contain condition to join *mid_table(2)* with *mid_table(3)*
...
tl(n-1).filter_expr contain condition to join *mid_table(n-1)* with *mid_table(n)*
tl(n).filter_expr contain condition to join *mid_table(n)* with *target_table.*

Each filter_expr can refer to columns in both linked tables prefixing by *s* columns in the first table and by *t* columns in the second. For example if

```
   tl(2).filter_expr=
s.full_name in (t.name || t.surname, t.surname || t.name)
```

then *full_name* is taken from *mid_table(2)* but *name* and *surname* columns from mid_table(3) table.

For data property maps linked tables are attached similarly as described above for object property maps. The difference is that there is no range class map for data property maps, therefore no *target_table* and no last link from *mid_table(n)* to *target_table*. Value expression for data property value (object part generation of RDF triples) can reference columns in all linked tables by appropriate aliases: *s* (or *t_domain*) for domain table, *mid1*, *mid2*, ... for intermediate tables.

Linked table usage for data property is shown below for far link example [2.3.2]

Table 28. OWL class maps to database tables for far link example

class_ map_ id	OWL class (rdf_id)	table_name	filter_expr	id_column_expr	instance_ uri_ prefix
1	Something	TABLE1		table1_id	Something

Table 29. OWL class instance RDF tripples for far ling example

Subject	Predicate	Object
<lumii #Something1>	<type>	<lumii#Something>
<lumii #Something2>	<type>	<lumii#Something >

The table below shows data property mappings in table *datatype_property_map* and referenced rows in *class_map, db_table* and *owl_datatype_property*.

Table 30. OWL datatype property class mappings far link example

class_ map_id	OWL_datatype_ property.rdf_id	table_name	column_expr	source_ column_expr	table_ link_id
1	localName	TABLE1	s.name		
1	farName	TABLE1	mid4.name	table2_id	1
2	farPath	TABLE1	s.name || ' ' || mid1.name || ' ' || mid2.name || ' ' || mid3.name	table2_id	

The table below shows table link information in rows of table *table_link* and referenced rows *(mid_table_name=mid_table_id→db_table.table_name)* .

Table 31. *table_link* table data for far link example

mid_ table_id	mid_table_name	source_ column_expr	target_ column_expr	next_table_link_id
1	TABLE2	table2_id	table3_id	2
2	TABLE3	table3_id	table4_id	3
3	TABLE4	table4_id		

Note a naming convention for table column aliases SQL expressions: by *s.* prefixing columns in table linked to domain class map of the property; by t. prefixing columns in table linked to range class map of the property; by *mid1, mid2, ...* prefixing column names in linked tables. In SQL commands that generate RDF triples prefix *s* is changed to *t_domain* and prefix *t* to *t_range*.

We have such SQL to generate RDF triples for *farPath* property values (simplified by not adding xsd:datatype information)

```
SELECT
  '<lumii#Something'
    + CAST(t_domain.STUDENT_ID AS varchar) + '>' as subject,
  '<lumii#farPath>' as predicate,
  t_range.name
    || ' ' || mid1.name
    || ' ' || mid2.name
    || ' ' || mid3.name
  as object
FROM
  TABLE1 t_domain
  INNER JOIN TABLE2 mid1
    ON t_domain.table2_id = mid1.table2_id
  INNER JOIN TABLE3 mid2
    ON mid2.table3_id = mid3.table3_id
  INNER JOIN TABLE4 mid3
    ON mid3.table4_id = mid4.table4_id
```

SQL for the other two properties are similar to this one (simpler expression for object part).

Table 32. OWL datatype property instance RDF triples for far link example

Subject	Predicate	Object
<lumii# Something1>	<lumii#localName>	table1 row1
<lumii# Something2>	<lumii#localName>	table1 row2
<lumii# Something1>	<lumii#farName>	table4 row1
<lumii# Something2>	<lumii#farName>	table4 row2
<lumii# Something1>	<lumii#forPath>	table1 row1 table2 row1 table3 row1 table4 row1
<lumii# Something2>	<lumii#farPath>	table1 row2 table2 row2 table3 row2 table4 row2

5.5 Mapping Validation

Since the mapping definition is stored in a RDB, validation is possible by using SQL. One can perform *Omission checks:* OWL classes or properties without corresponding class or property maps, DB tables not used in any class maps, DB columns not used in any column expression. *Consistency checks* may be used for property_map-to-class_map relation correspondence to property-to-class domain/range relation. The results of these checks are to be evaluated to decide, whether an error has been found, or the irregularity is by the mapping design. After

loading the data into the target RDF data store, further checks of ontology data-to-schema consistency can be performed by means of SPARQL queries or invoking reasoners such as Pellet[7] or FaCT++ [8].

For example, a simple SQL returns a list of OWL classes without class maps:

```
SELECT rdf_id AS class FROM owl_class c
WHERE NOT EXISTS
(
  SELECT 1 FROM class_map cm WHERE c.owl_class_id=cm.owl_class_id
) AND c.is_abstract=0
```

The following SQL finds all OWL datatype properties that are without property maps:

```
SELECT
  c.rdf_id AS domain,
  dp.rdf_id AS owl_datatype_property
FROM owl_datatype_property dp
  INNER JOIN owl_class c ON dp.domain_id=c.owl_class_id AND
c.is_abstract=0
 WHERE NOT EXISTS
 (
   SELECT1 FROM datatype_property_map dpm
   WHERE dpm.owl_datatype_property_id=dp.owl_datatype_property_id
 )
```

6 A Latvian Medicine Registries: A Case Study

We transformed Latvian medical registry data from relational databases into RDF triples corresponding to medicine ontology. The data mapped from RDB to OWL format consisted of 6 Latvian medical registries (Sugar registry, Multiple sclerosis registry, Injury registry, Mental illness registry, Cancer registry and Narcotic registry) [11],[12], including 106 source database tables, 1353 table columns and total more than 3 million rows, altogether 3 GB of data. The corresponding OWL ontology had 172 OWL classes, 814 OWL data properties and 218 OWL object properties.

The mapping has been run in February 2010 on a HP ProBook 4710s laptop computer with Intel Mobile Core 2 Duo T6570 2.1GHz processor and 3 GB of RAM running 32-bit Windows Vista operating system. Microsoft SQL Server 2005 served the mapping DB as well as source DB (Medicine DB). The triple generation process from source DBs produced about 42.8 million triples and it has taken 18.5 minutes, out of which 6.5 minutes for raw triple generation and storage into relational tables, 8 minutes for indexing of triple tables, 4 minutes for *ClassConstraint* enforcement and 6 minutes for triple exporting from table to text files in N-TRIPLE format (total file size 3.4GB).

We executed the process following the schema depicted in **Figure 49**. SQL commands were generated from mapping data stored in tables of RDB2OWL mapping schema (depicted in **Figure 50**): *class_map, object_property_map, ...* Execution of these generated SQL commands in the source database generated the target RDF triples. The tables below shows the element counts and timing details.

Table 33. Element counts in target ontology

Item	Count
OWL Classes (non abstract)	168
OWL data properties (domain non abstract)	810
OWL object properties (domain and range non abstract)	198
Total objects	1176

Table 34. Mapping counts in RDB2OWL database

Item	Count
Class_map rows	170
Datatype_property_map rows	832
Object_property_map rows	220
Total objects	1222

Migration done 100%

Table 35. Generated SQL counts for RDF generation

Item	Count
SQL for OWL class instances	169
SQL for OWL data property instances	832

SQL for OWL object property instances	220
Total objects	1221

Table 36. Source DB (database file size: 3G)

Item	Count
Tables	106
Table Columns	1353
Total rows in all tables	3 054 618

Table 37. Generated triples

Item	Count
Class instance count	5411395-32084=5 379 311
Data property instance count	17 953 290
Object property instance count	19 509 296
Total	42 841 897

Table 38. Timings for triple generation

Step	Time (min:sec)
RDF triple generation (42,84 millions)	6:30
Triple indexing	8:05
Deletion of OWL class instance triples without required properties (deleted 32084 of 5411395 or 0,6%)	3:55
Export to RDF dump file	6:18
Total time	24:48

After storing triples (indexed) in source DB its DF file size grew from 3G to 7G.

We can compare our timings with another approach. The same Latvian medicine registries use case was tried for mapping approach by general purpose graphical model transformation language [36, 37] (described in Section 3.2.9) where the same RDB-to-RDF migration took 90 minutes and the improved version (with MOLA-to-SQL compilation) was applied for one of 6 medicine registries and executed twice faster. When implemented for all 6 registries, expected time would be 45 minutes. The comparison does not take into consideration hardware parameters used for migration.

In this practical RDB2OWL application to medicine case, there were many occurrences when a class map mapped several OWL classes to one source database table: such "splitted" tables were 17 out of 106 tables and 76 classes of 172 in total. This means that 44% of all OWL classes were in one-to-many relation from tables to classes and 56% in one-to-one relation.

Table 39. Database tables mapped to more than one OWL class

Source database table	OWL class count

Source database table	OWL class count
MSMSAnamneze	10
MSSudzibas	9
MSKurtzke	9
PSIHKlasifikators	6
CDUzskaitesKarte	6
IDBIevainojums	5
MSKlasifikators	5
IDBIevainojumsDetalas	4
NARKKlasifikators	4
VRKlasifikators	4
VRKlasifikatorsKods	3
VRUzskaitesKarte	3
MSDzivesAnamneze	2
IDBKlasifikators	2
XNonemtsNoUzskaites	2
XpersonasKarte	2

We annotated Latvian medical registry ontology in RDB2OWL Core plus language. We marked by `?Out` decoration 54 class map expressions (of 172) were to specify multiclass conceptualization usage. These classes had 542 out of 814 OWL data properties "outgoing" from them (marked classes as domain for the properties). It means save of effort in 542 times explicit writing SQL condition like `"AND some_field NOT NULL"` and shows that for Latvian medical registry there is benefit of using multiclass conceptualization of RDB2OWL language described in Section 4.3.1.

In numerous situations many year values were stored in one varchar type field value (e.g., '199920012005') but corresponding data property having separate instances for each value {'1999', '2001', '2005'}. The value splitting can be implemented by joining the source table with auxiliary table *Numbers* having single integer type column filled with values from 1 to 999 (see [76]), as in the function: *split4(@X)=((Numbers;len(@X)>=N*4).substring(@X,N*4,4)),* The application *split4(FieldX)* then splits character string into set of substrings of length 4. The definition of function *split4* was added as annotation to OWL ontology itself and used (called) it from 111 out of 814 data property maps.

Another benefit from using functions was in dealing with the differences on how Boolean values are specified: in databases numerical values 0 and 1 are typically used to represent true and false. For OWL data properties with range *xsd:boolean*, the literal values: *false^^xsd:boolean* and *false^^xsd:boolean* should be used. To avoid numerous usage of low level SQL coding for value transformation

```
CASE WHEN field=1 THEN
  'true^^xsd:boolean'
ELSE
  'false^^xsd:boolean'
END
```

a function *boolT* was used (the definition use RDB2OWL built-in function #iif, see **Figure 52** below) The *boolT* function was used in 229 out of 814 data property map descriptions. There are other functions defined as well: *isEqual* to specify Boolean value on equality conditions of two data values (e.g., table fields). Function *isOneOf2*

helps in situations when condition for class maps or property maps needs to specify that some value (table field) is in list of 2 values- several such cases in Medical registries.

For object property maps there were many cases when default foreign-to-primary or primary-to-foreign key relation could be used: 83 object property map expressions was simple "->" and 10 "=>" out of total 218. It means that almost half of all object property map descriptions benefited from RDB MM awareness in RDB2OWL Core language. There were also object property maps that used only key information partly- only in one end. An example is object property map description "*[SmagumaPakapeID]->"* for property *apdApdegumaPakape* in figure below.

We used named references in 26 cases (see name *TraumasDetala1* and *TraumasDetala2* in **Figure 52** below that were defined in two class map descriptions to the *TraumasDetala* class and used in respective object property maps).

Figure 52 and **Figure 53** below shows fragments of annotated ontology for Injury and Multiple sclerosis registries. We are presenting the fragments with most typical mapping situations found in the migration specification. The ontology annotations given here are sufficient for the entire mapping specification; clearly, for the presentation to the end user, they should be enriched with user-readable comments.

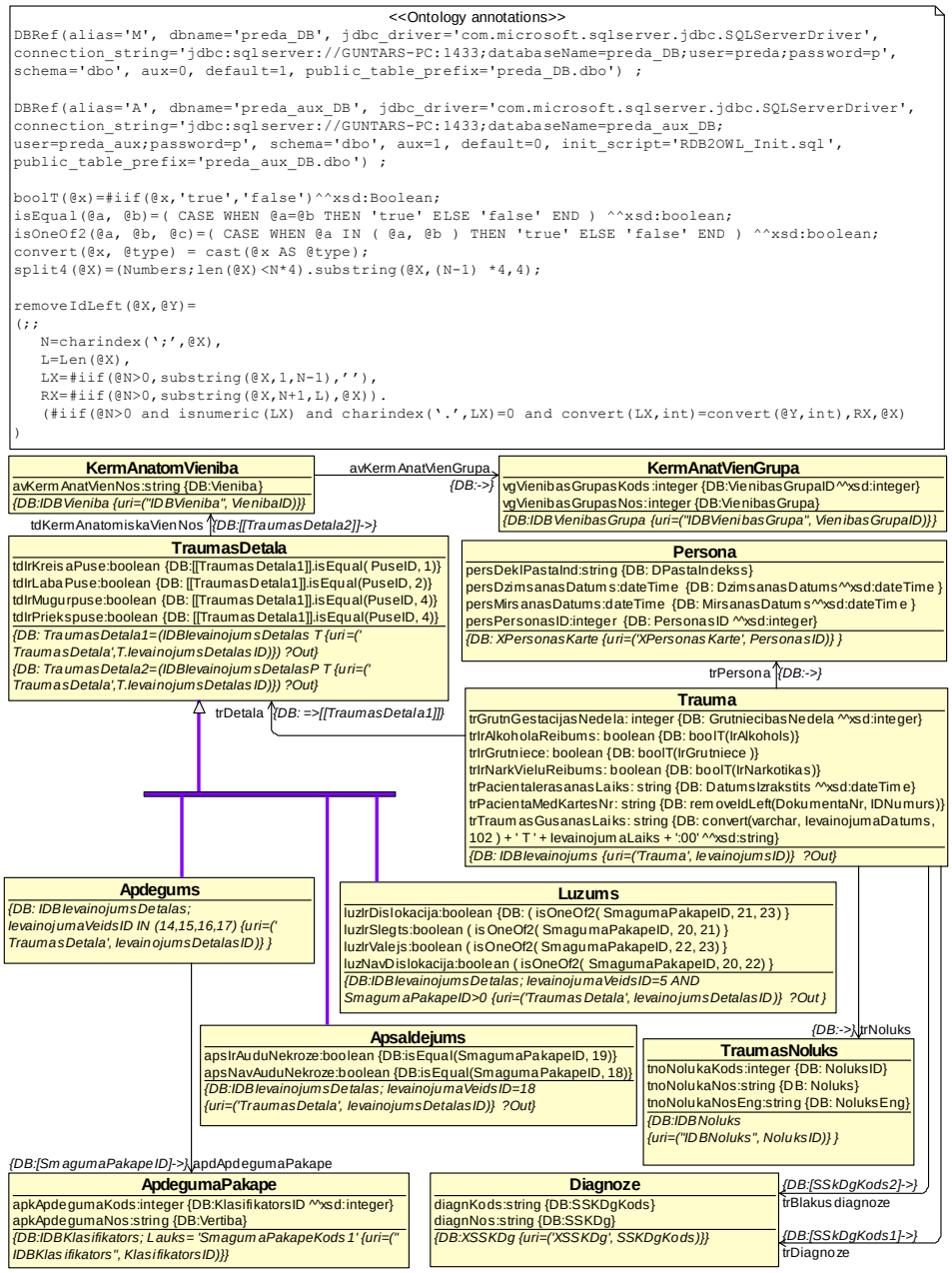


Figure 52. Annotated medicine ontology using RDB2OWL Core language: fragment of Injury registry

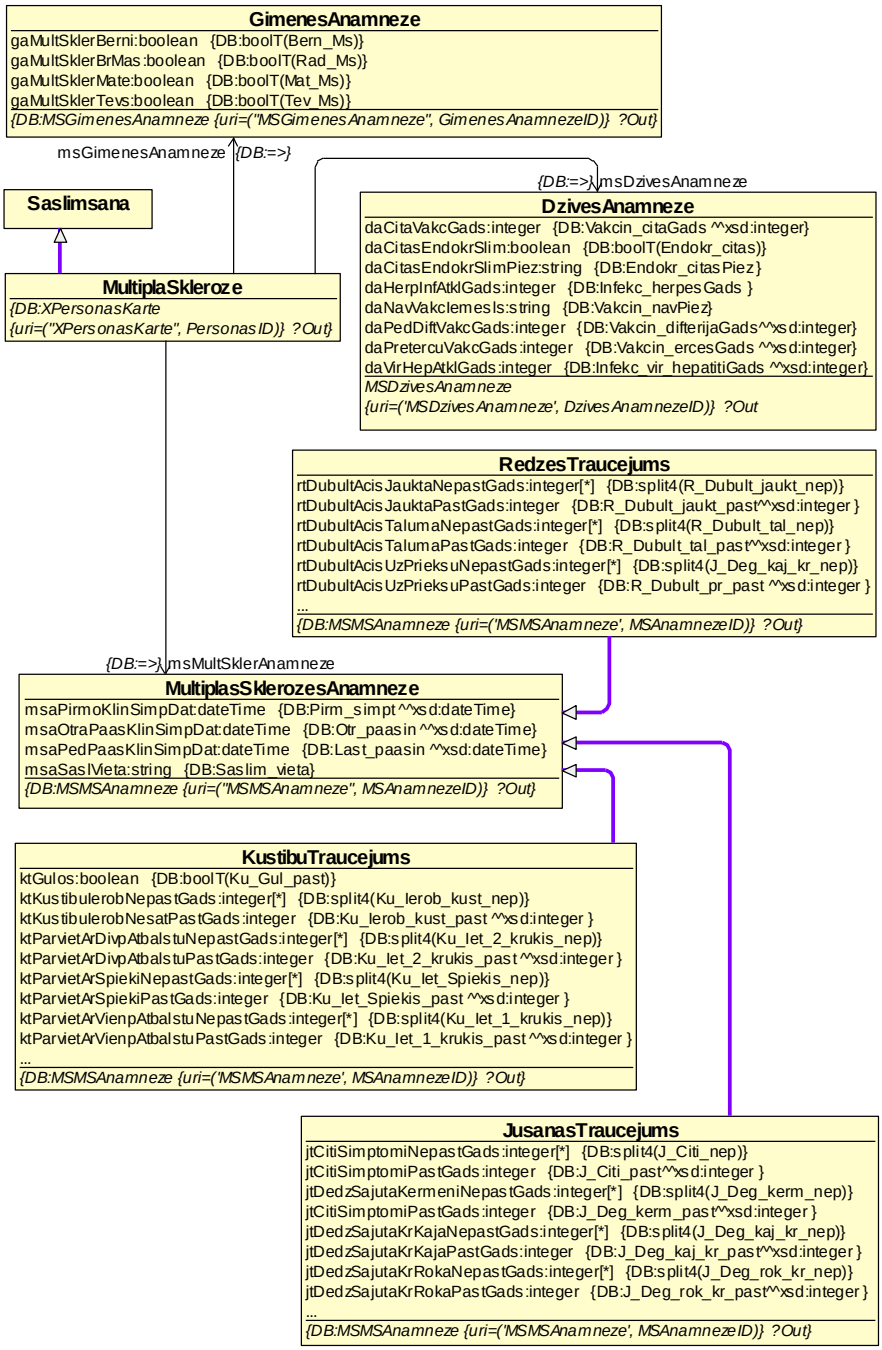


Figure 53. Annotated medicine ontology using RDB2OWL Core language: fragment of Injury registry

7 Implementation for RDB2OWL mapping specification language

7.1 Overall implementation architecture for RDB2OWL language

Figure 54 shows the total implementation schema of RDB2OWL language. We briefly describe the main process steps of multistep transformation process leading from annotated ontology file to the RDB2OWL mapping schema from which RDF triples can be generated as described in sections 5.1 and 5.2.

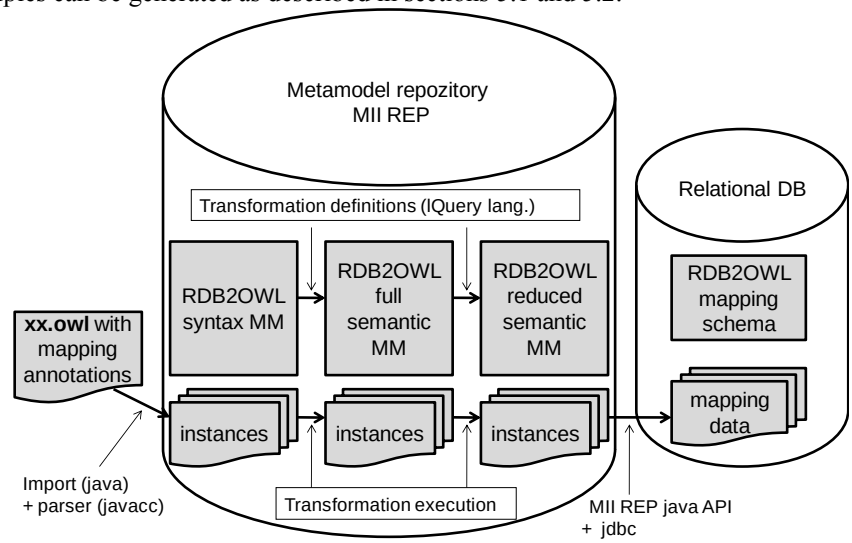


Figure 54. Implementation architecture of RDB2OWL language

The process starts by reading the target ontology annotated with RDB2OWL mapping language. The annotations are parsed (javacc [75]) into RDB2OWL syntax metamodel (shown in **Figure 55**) instances in repository. These instances correspond to mere syntactic structure of mapping expressions. Then these instances are analyzed to add the missing information. Example of missing information: expression *credits=amount+100* contain literal '*amount*' but it is not clear if it stands for defined variable name, function name or database table column name.

After the syntax parsing completion, another transformation steps are started that analyses what instances are created and creates additional instances or links for semantic information. For example if '*amount'* is not found as defined function name or variable name but is found to be table column name from table context the this literal stands for column and this information is recorded. The omissions are also filled, for example explicit columns name omissions in navigation links. At the end of

semantic analysis, RBB2OWL semantic MM instances are obtained. At this point metamodel instances correspond to RDB2OWL Core Plus language. Then another transformation step converts the metamodel instances from RDB2OWL Core Plus level into what we call RDB2OWL reduced semantic MM level: changes high level constructs into lower, for example, changes function calls into basic expressions (merging caller table expression with table expression of the called function). The last step is to transform the obtained RDB2OWL metamodel instances into RDB2OWL mapping schema data, from which RDB2OWL triples can be generated.

The detail grammar of RDB2OWL language written in ALTLRWorks tool is given in Appendix A.7. Its parser java implementation in JavaCC (Java Compiler Compiler JavaCC - The Java Parser Generator) [75] generates instances of RDB2OWL full syntactic metamodel shown in **Figure 55** by using MII REP repository [10] and its java API.

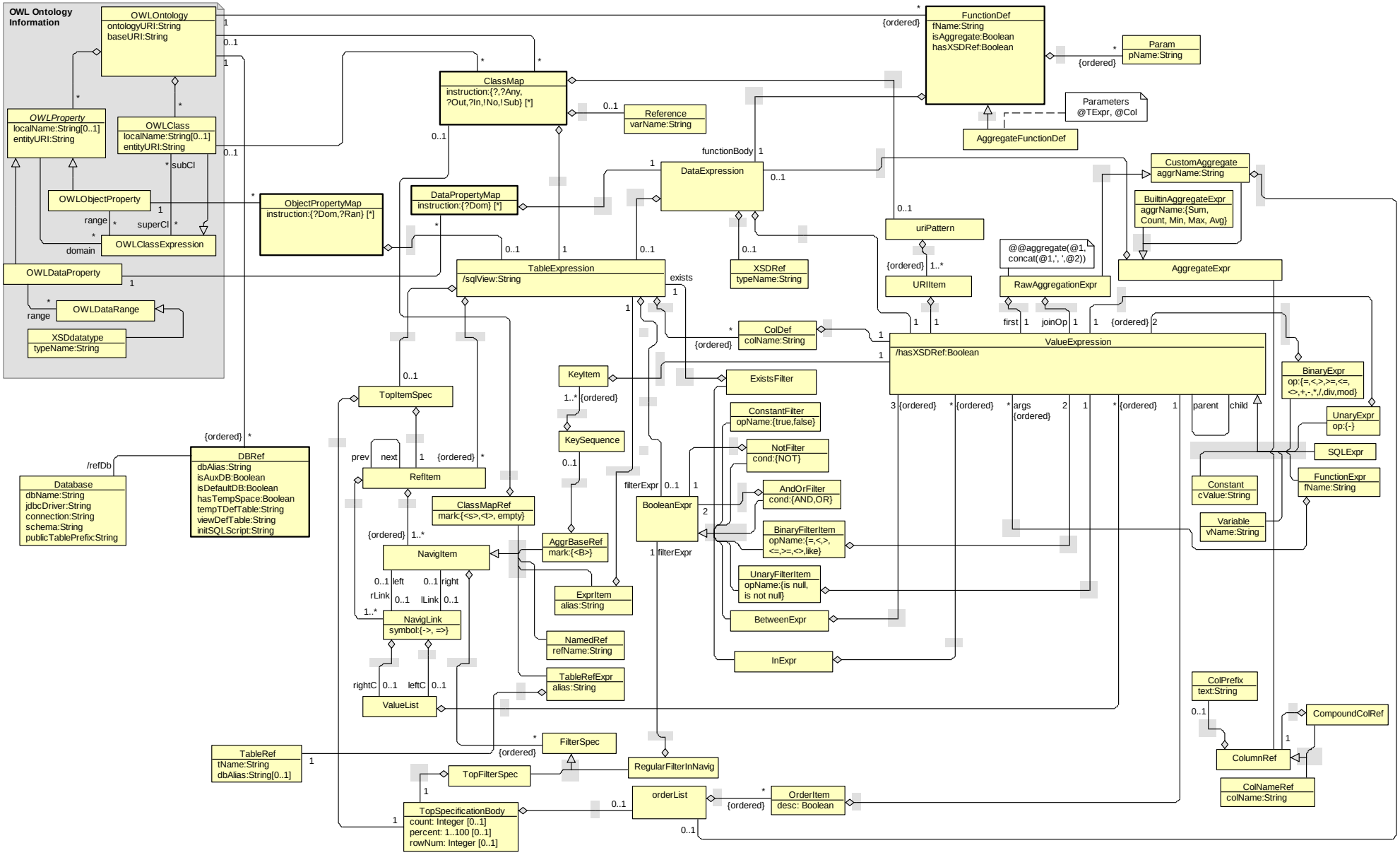


Figure 55. RDB2OWL full syntactic metamodel

The full RDB2OWL language metamodel is packed in Figure 56.

The instance data for associations and classes displayed in solid lines (comprising full syntax metamodel) are obtained by parsing of RDB2OWL annotations (using javacc). Instances for classes and associations denoted by dotted lines (comprising full semantic metamodel of RDB2OWL Core Plus language) are calculated by processing instances of full syntax metamodel generated by syntax parsing before.

NamedRef instances are split into *DevVarRef* or *ClassMap* instances depending on weather *refName* attribute value is found as *Reference.varName* attribute value or *OWLClass.localName* attribute value.

There are various *ref* links (*ClassRef* → *OWLClass, TableRef*→*Table, TableColname*→*Table*, etc) that can be calculated during semantic analysis after full syntax metamodel is filled with instances by grammatical parsing.

LQuery and lua language code listings for various semantic transformation steps are given in Apendix A.8.

All options of RDB2OWL language (Raw, Core and Core Plus levels) described in sections 4.1-4.3 are combined in metamodel of Figure 56 with additional classes for implementation purposes.

Classes and associations denoted by solid lines represent syntactic structural information of mapping expressions and source RDB schema and target OWL ontology. The instance data for these classes and associations can be filled by parsing RDB2OWL mapping expressions and analyzing RDB schema and OWL ontology.

Classes and associations denoted by dotted lines are for further semantic information not contained directly in the mapping expressions.

For example, object property map expression

```
<s> => REGISTRATION -> <t>
```

for object property *takes* is parsed into three instances of subclasses of *NavigItem* class and other classes: *ClassMapRef* for "<s>", *TableRefExpr* with linked *TableRef* for "REGISTRATION" and *ClassMapRef* for "<t>". Link *ref* between *ClassMapRef* and *ClassMap* instances is calculaled by analysing ontology structure (get the only *ClassMap* instance that is linked to domain class for the corresponding OWL property. Link *ref* between *TableRef* and *Table* classes is created by finding *Table* instance for which *Table.tName=TableRef.tName.*

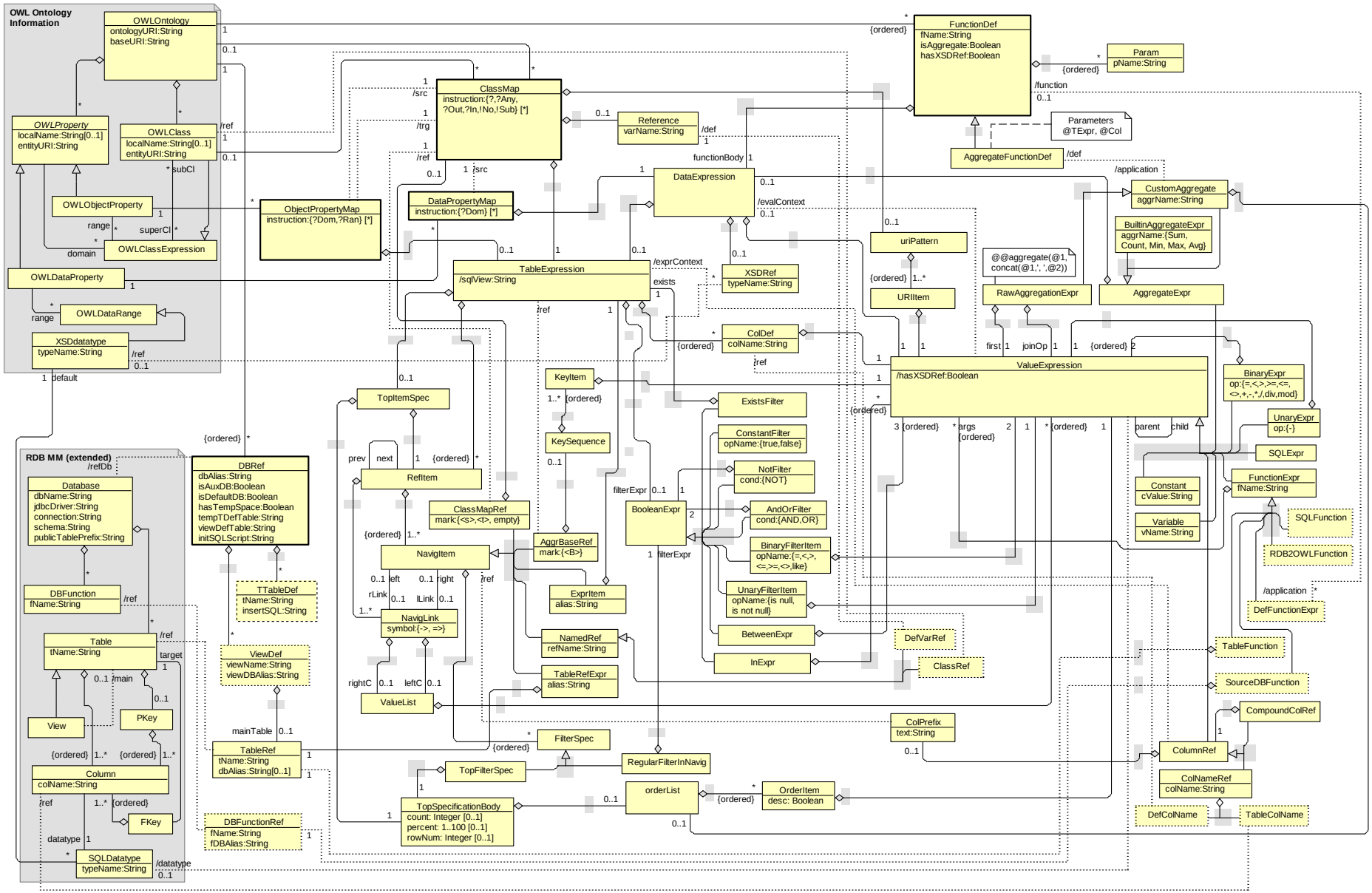


Figure 56. RDB2OWL full semantic metamodel

7.2 Transformation steps

Syntax parsing into instances of syntax metamodel.

The RDB2OWL grammar (see Appendix A.7) parser implementation in javacc [75] processes all RDB2OWL annotations attached to the target OWL ontology and creates instances of RDB2OWL full syntax metamodel shown in **Figure 55**. For example, class map expressions attached to OWL classes transformed to instances of *ClassMap* with links to corresponding instances of *OWLClass* and *TableExpression* and *References* classes. Javacc code sample fragment (action code in italic):

```
PARSER_BEGIN(Rdb2Owl)
import lv.lu.mii.repository.KPRepositoryPlus;
…
public class Rdb2Owl {
  private KPRepositoryPlus rep=null;
  …
  TOKEN : {< CDecoration   : "?" | "?Out" | "?In" | "!NoMap"
                           |   "!SubClean" >}
  …
  long classMap() :
  {
      String name;
      Token t;
      long classMapId;
      long uriPatternId;
      long tableExprId;
  }
  {
          {classMapId=mappingProcessor.createClassMap(owlClassName);}
  (
     LOOKAHEAD(2)
     name=defName()
     "="
               {mappingProcessor.createReference(classMapId, name );
                        }
  )?
  tableExprId=tableExpr()
              {rep.associateObjects(
                 "tableExpression",
                 classMapId, tableExprId) ;
              }
  (
    uriPatternId=uriPattern()
             {rep.associateObjects(
               "URIPattern", classMapId, uriPatternId);
             }
  )?
(
   t=<CDecoration>
            { rep.setObjectAttribute(
                classMapId, "instruction", t.image);
            }
    )*
            {return classMapId; }
  }
```

```
long tableExpr():
{
   long id=rep.createObject("TableExpression");
   …
{
{
  …
}
```

linkObjectPropertyMapsToClassMaps.lua (Appendix [A.8.3])

Links *src* and *trg* are created (used for class map reference marks <s> and <t>) from *ObjectPropertyMap* to *ClassMap* in following way. For each *OWLObjectProperty* instance *p*, find the corresponding domain and range classes, and then the sole class map attached to each. These are the targets of *src* and *trg* links from *p*. Similarly, script *linkDataPropertyMapsToClassMaps.lua* (Appendix [A.8.4]) creates *src* links.

splitNamedRef2SubClasses.lua (Appendix [A.8.5])

NamedRef instance is split into instances of one of its subclass *DefVarRef* or *ClassRef* by analyzing if *refName* attribute value is found as name of defined variable or class name: as equal to *OWLClass.localName* or *Reference.varName*. For example, annotated mini-University ontology (**Figure 34**) has class map:

```
T=Teacher {uri=('Person_Id',IDCode)}
```

and an object property map:

```
[[Teacher]][teacher_id]->[[T]]
```

In this case, the syntax parser at first would transform both the *[[Teacher]]* and *[[T]]* into instance of the *NamedRef* class. Then the first goes to *ClassRef* and the second to *DefVarRef* instance because there is class named *Teacher* and defined variable named *T*.

linkColPrefix2NavigItem.lua (Appendix A.8.6)

Links *ref* are created from *ColPrefix* to corresponding *NavigItem* instance. This linking is required to associate column prefixes used in value expressions (e.g., filter expressions in table expressions) with corresponding navigation or reference item. For example, for table expression

```
Teacher t, Course c; t.teacher_id=c.course.id
```

t from *t.teacher_id* is parsed into *ColPrefix* instance and *Teacher t* is parser into *TableRefExpr* (a subclass of *NavigItem*) with *alias='Teacher'*. These two instances of *ColPrefix* and *TableRefExpr* are linked.

changeEmptyItemsToClassMapRefs.lua (Appendix A.8.7)

Changes empty *ClassMapRef* instances (mark="empty") to non empty *ClassMapRef* instances (mark="<s>" or mark="<t>"). If the first *NavigItem* in the list (as *RefItem→{ordered}NavigItem*} mark it "<s>" and the last empty *NavigItem* change to *ClassMapRef* with mark="<t>". For example, the transformation means table expression

```
=>Registration->
```

change to

```
<s>=>Registration-><t>
```

linkClassMapRef2ClassMap.lua (Appendix A.8.8)
The *ref* links are created from *ClassMapRef* to *ClassMap* determined as follows. The ClassMapRef with mark="<s>" or "<t>" as NavigItem is part of *RefItem* and that of *TableExpression* which is assigned to some property map instance *pm* (of *ObjectPropertyMap* or *DataPropertyMap*). *ClassMap* instances linked to *pm* by *src* or *trg* links are the ones used for *ref* link.

linkXSDRef2XSDDatatype.lua (Appendix A.8.9)
Creates *ref* links from *XSDRef* to *XSDDatatype* based on *typeName* attribute value.

splitColNameRef2Subclasses.lua (Appendix A.8.10)
Splits each ColNameRef class instance into instance of one of its subclasses: DefColName or TableColName. Decision which subclass is taken is resolved by finding if colName property value is found as ColDef.colName property value attached to instance of TableExpression from the context of the value expression (is exprContext link target). If ColDef is founf then ColNameRef is split into instance of DefColName otherwise of TableColName. As an example, let us take two equivalent data property map expressions:

```
- (TEACHER t).( t.level_code || ' ' || t.name)
- (TEACHER t; fullName=t.level_code || ' ' || t.name).fullName
```

In the firsts example *level_code* (in bold) is parsed first into *ColNameRef* and then split into *TableColName*. In the second expression, the last (bold) *fullName* is finally split into *DefColName*.

fillExpliciteNavigationColumns.lua (Appendix A.8.11)
Fills explicit column expressions into navigation links they are omitted. For example, object property map expression for *teaches* property

```
TEACHER=>COURSE
```

is transformed into

```
TEACHER[teacher_id]=>[teacher_id]COURSE
```

The transformation considers the source database schema structure information (primary/foreign key columns) that is stored in metamodel (RDB MM (extended) part of **Figure 56**). Link arrow =>means Primary-to-Foreign key link and
-> denotes foreign-to-primary key link. The transformation creates instances of *ValueList* for *rightC and leftC* link ends.

linkObjectPropertyMapsToClassMapsByNamedRefs.lua (Appendix A.8.12)
The *src* and *trg* links are created from *ObjectPropertyMap* to *ClassMap* instances when they are not created by default algorithm: (unique class map of domain/range class of the property). The *src* class map of object property map is determined as follows:
1) let *x* be the first *NavigItem* instance of the first *RefItem* instance;
2) if *x* is of type *NamedRef* then *src* class map is *referencedClassMap(x)*
The *trg* class map of object property map is determined as follows:
1) let *x* be the last *NavigItem* instance of the last *RefItem* instance,
2) if *x* is of type *NamedRef* then *trg* class map is *referencedClassMap(x);*
The *referencedClassMap(x)* is determined as follows:
1) if *x* is of type *ClassRef* and *c* is *OWLclass* instance that is linked to *x* by *ref* link

then return the only class map that is ascribed to class *c;*

2) if *x* of of type *DefVarRef* then return class map linked to *x*:

x -->(def) --> ClassMap.

For example, this script creates *src* and *trg* links for two object property maps for the property *PersonID* (mini university example):

2 class maps of *PersonID* class:

classMap1: `S=Student {uri=('PersonID', IDCode)}`

classMap2: `T=Teacher {uri=('PersonID', IDCode)}`

2 property maps of the personID object property:

propMap1: `[[Student]][student_id]->[[S]]`

propMap2: `[[Teacher]][teacher_id]->[[T]]`

In this example *src* and *trg* links created are:

propMap1 -->(src) The only class map of Student class

propMap1 -->(trg) classMap1

propMap2 -->(src) The only class map of Teacher class

propMap1 -->(trg) classMap2

Similarly, *DataPropertyMaps* are linked to *ClassMaps* by named refs (see linkDataPropertyMapsToClassMapsByNamedRefs.lua in Appendix A.8.13)

create_refLinks2TableAndColumn.lua (Appendix A.8.15)

The *ref* links are created from *TableRef* to *Table* classes. The target *Table instance* is determined from database alias and table name stored as *tName* and *dbAlias* attribute values. For example, in table expression

```
PERSON p, A.ADDRESS adr; p.addr_code=adr.code
```

the reference item `person p` is parsed by grammar into *TableRefExpr* and linked *TableRef* instance with *tName="PERSON"* and *dbAlias=null*. The *Table* instance with such a name is found for default database (*DBRef.isDefault=true*). The reference item `A.ADDRESS adr` is parsed into *TableRef* instance with *tName="ADDRESS"* and *dbAlias="A"*. The *Table* instance is found for the specified different database.

insertDefaultURIPatterns.lua (Appendix A.8.16)

If Uri pattern is not explicitly specified as part of class map then it is inserted following with the default pattern: the name of the main table (for the first navigation item for the first reference item) concatenated with values of all primary key columns. For example, the simple table expression for *MandatoryCourse* class

```
COURSE; required=1
```

is supplemented with default URI pattern

```
COURSE; required=1 {uri=('COURSE', course_id)}
```

This transformation creates *URIPattern* and list of *URIItem* with *ValueExpression* for each part (of type *Constant* with *cValue="COURSE"*and *TableColName* for *course_id* column).

7.3 Client language implementation as Java application

We implemented client language processing as java application that takes as input annotated OWL ontology and after several steps fills table data in RDB2OWL

mapping DB schema described in Section 5.2 from which RDF triples can be generated implemented as another java application described in Section [7.4]. At the time of writing implementation is done for subset of RDB2OWL Raw and Core language. We implemented the basic language constructs and successfully performed RDF triple generation from annotated mini-university ontology with mappings in RDB2OWL Core language.

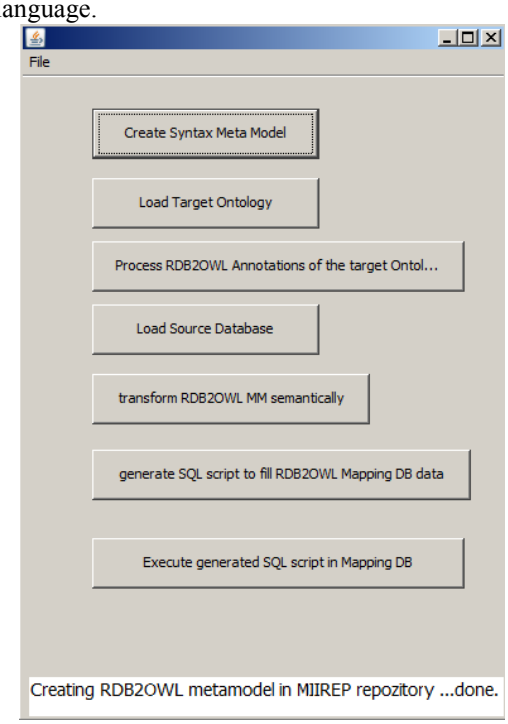


Figure 57. Main form of RDB2OWL client language application

We will briefly describe what transformation steps are executed that lead from annotated ontology to mapping DB data. The steps correspond to button in the main form of the application.

<Create Syntax Meta Model> creates RDB2OWL metamodel of **Figure 56** in LUMII REP [10] repository for MOF style metamodel.

<Load Target Ontology> processes OWL ontology by java OWL API [77] to fill RDB2OWL metamodel with information about structure of target ontology. Instances are created for respective classes in upper left part of **Figure 56**: Ontology, OWLClass, OWLObjectProperty, etc

<Process RDB2OWL Annotations of the target ontology> gets all RDB2OWL expressions: class maps attached as annotation to OWL classes, property maps as annotation attached to OWL properties and source database description *DBRef* (syntax shown in **Figure 42**) as annotation attached to OWL ontology itself. All RDB2OWL mapping expressions retrieved from ontology annotations are processible by parser of our RDB2OWL language grammar (see Appendix A.7) implemented in javacc [75]. Parsed RDB2OWL annotations are transformed into instances of

RDB2OWL metamodel by using LUMII REP java API in action codes. Only syntactic information is written in metamodel. For example, for navigation item "*Teacher c*" no *ref* link is created from *TableRef* to *Table* due to missing information at this point- structure of source database structure is loaded in the next step.

<Load Source Database> processes source database schema metadata by java jdbc API to fill RDB2OWL metamodel with information about structure of source database. Instances are created for respective classes in lower left part of **Figure 56**: *Database*, *Table*, *Column*, *Pkey*, *Fkey*. We take Database description (connection information) from *Database* and *DBRef* (filled previously from *DBRef* expressions in ontology annotation).

<Transform RDB2OWL MM semantically> RDB2OWL metamodel instances contain at this point only syntactic information and further processing is needed to add semantic information: splitting *NamedRef* objects into *DefVarRef* and *ClassRef* with *ref* links to *Reference* or *OWLClass* respectively, creating *scr* and *trg* links from *ObjectPropertyMap* to *ClassMap*, creating *ref* link from *TableRef* to *Table* and *ref* link from *TableColName* to *Column* and many other. All these transformations are implemented as lQuery [78] and lua [79] scripts shown in Appendix A.8.

<Generate SQL script to fill RDB2OWL Mapping DB data> RDB2OWL metamodel instances that now contain semantic information are processed by lQuery [78] and lua [79] scripts to generate SQL insert statements to fill RDB2OWL Mapping DB schema tables.

<Execute generated SQL script in Mapping DB> executes generated SQL statements generated in previous step to fill RDB2OWL Mapping DB schema tables.

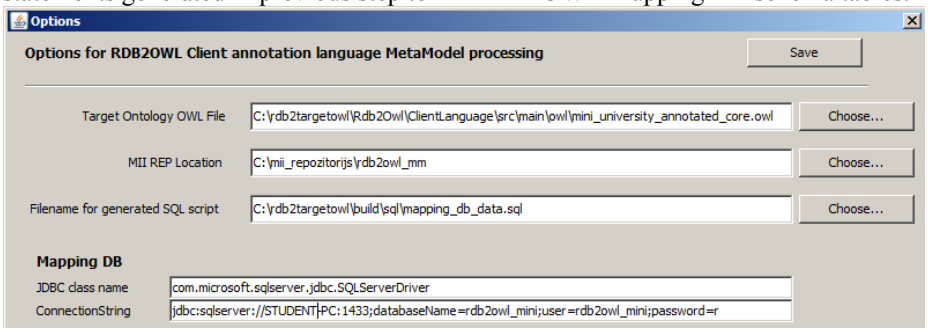


Figure 58. Options form of RDB2OWL client language application

Options form allows to specify necessary parameters: annotated target ontology, LUMII REP repository folder location, generated SQL script file name and connection information to RDB2OWL Mapping DB.

7.4 Java application for RDF triple generation from mapping DB data

The RDF triple generation two-step processes (**Figure 49**) we implemented as UI java application. When connection information to the mapping and source databases are entered then both processes can be executed by pressing corresponding command

button. Below is shown the execution screenshot when button <Generate Target RDF triples> was pressed (therefore disabled)

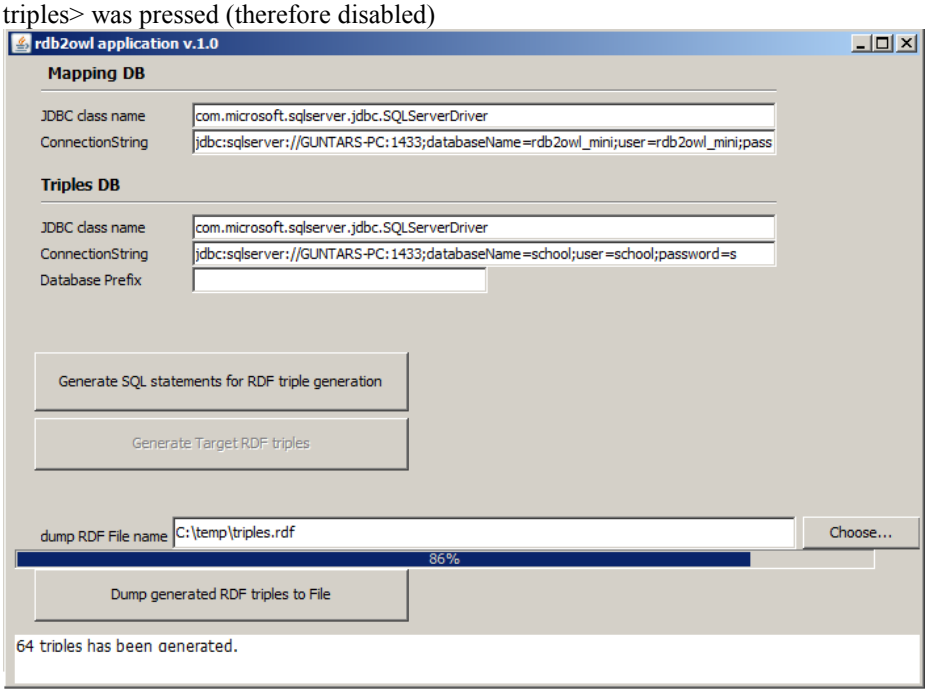


Figure 59. Application form for RDF triple generation

<Generate SQL statements for triple generation> executes SQL scripts shown in Apendix A.6 that generate a set of SQL insert statements to be executed in source database to generate RDF triples for target ontology.

Currently scripts from Apendices A.6.3,A.6.4 and A.6.5 are each designed for specific number of *table_link* usage (specific lengts of table link chain). Additional *table_link* usages can be easily added or even it is possible to write code that generates SQL for property triple generation for any given *table_link* usage count. For real life application rare are cases when more that 5 tables are joined in link. There is another problem when one SQL script processes Mapping DB data: not all database vendors use standard SQL language. For example Microsoft SQL Server uses + operator for string concatenation while Oracle DB and many others use SQL standard operator ||. This means that it is not possible to use one SQL for diffenent databases.Therefore a better approach is to use platform and database independent java jdbc API to process Mapping DB data because generally database vendors provide jdbc driver.

<Generate target RDF triples> executes generated SQL insert statements generated in previous step to generate RDF triples that are stored in database tables in database specified in text fields under *Triples DB* label.

<Dump generated RDF triples to File> allows to save RDF triples in text file in .n3 serializstion.

8 Conclusions

We have presented RDB2OWL approach to RDB-to-RDF/OWL mapping specification that re-uses the ontology structure as the backbone of the mapping specification by putting the mapping definitions in the OWL ontology entity annotations.

As the mapping examples show, the approach can be used for a convenient mapping definition. Combining the power of the RDB2OWL mapping definition approach with visual ontology modeling means such as OWLGrEd [25,26] notation and editor can be a viable mechanism for the RDB semantic re-engineering task. Since the ontology annotation mechanism is a part of ontology definition, it means that the RDB2OWL-annotated ontologies can be used also outside the concrete ontology editor.

The RDB2OWL approach has been successfully used for a "real-size" task of semantic re-engineering of databases in Latvian medical domain. There is work in progress towards the implementation of the full set of RDB2OWL constructs, including RDB2OWL parsing on a concrete syntax level and integrating into OWLGrEd editor.

It seems to be a plausible and interesting task to adapt the mapping constructions considered here also for RDF/OWL-to-RDF/OWL mapping definition that may be useful in transformation from the "technical data ontology" to the conceptual one, after the initial data – be these in RDB or some other format – have been exposed to the RDF format using a straightforward and technical structure preserving embedding.

The main goal of the work was to make relational databases accessible to semantic web technologies, and particularly, accomplished by mappings between RDB and RDF/OWL. The main results are:

- RDB-to-RDF/OWL mapping language RDB2OWL was designed which is oriented to be readable by humans, concise. The development of the language proves that RDB-to-RDF/OWL mapping language with high-level constructs (e.g., user-defined functions, multiclass conceptualization, and implicit use of information from source database and target ontology) is possible.
- A syntax parser for the RDB2OWL mapping language was created.
- A subset of RDB2OWL mapping language implementation complemented with user interface is created. With its help the RDB-to-RDF/OWL mapping data from annotated ontology can be transformed into relational table data (in RDB2OWL mapping schema) from which RDF triple generation is possible.
- Ontology of Latvia Medicine registries was annotated with RDB2OWL mappings language expressions showing that the language is applicable to practical industry use case, expressing the correspondence between relational database and domain ontology.
- RDB2OWL mapping implementation was developed (execution environment) where the RDB-to-RDF/OWL mapping information is stored in relational database schema and RDF triples are generated by

SQL based processes. For these processes a user interface application was developed.

- RDB2OWL mapping implementation was applied to semantic re-engineering of Latvia Medicine 6 registry databases where 42 million triples were generated in 18,5 minutes (without dump export to text file).

The development of RDB2OWL mapping language is in progress, enlarging the set of implemented constructs. One of future perspectives is to create compiler from RDB2OWL into the new W3C standard language R2RML. Then RDB2OWL could be used as convenient language to define mappings and triple execution could be delegated to R2RML supporting tools. Currently there are triple generation support for D2RQ, Virtuoso RDF Views and Revelytix RDB mapping languages, and it could be worth considering to build compiler from RDB2OWL to these languages.

9 References

1. Tim Berners-Lee, James Hendler and Ora Lassila, "The Semantic Web", Scientific American, May 2001, p. 29-37.
2. Resource Description Framework (RDF), http://www.w3.org/RDF/
3. RDF Vocabulary Description Language: RDF Schema, http://www.w3.org/TR/rdf-schema/
4. Web Ontology Language (OWL), Document Overview, http://www.w3.org/TR/owl-overview/
5. Web Ontology Language (OWL), Structural Specification and Functional-Style Syntax http://www.w3.org/TR/owl2-syntax/
6. W3C RDF Validation Service http://www.w3.org/RDF/Validator/
7. Pellet, reasoner http://clarkparsia.com/pellet
8. FaCT++, reasoner, http://owl.man.ac.uk/factplusplus/
9. W3C SWEO Linking Open Data community project URL: http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
10. J.Barzdins, G.Barzdins, R.Balodis, K.Cerans, et.al.: (2006). Towards Semantic Latvia. In Communications of 7th International Baltic Conference on Databases and Information Systems, pp.203-218.
11. G.Barzdins, E.Liepins, M.Veilande, M.Zviedris: Semantic Latvia Approach in the Medical Domain. Proc. 8th International Baltic Conference on Databases and Information Systems. H.M.Haav, A.Kalja (eds.) Tallinn University of Technology Press, pp. 89-102. (2008).
12. G.Barzdins, S.Rikacovs, M.Veilande, and M.Zviedris: Ontological Re-engineering of Medical Databases, Proceedings of the Latvian Academy of Sciences. Section B, Vol. 63 (2009), No. 4/5 (663/664), pp. 20–30.
13. J.Barzdins, G.Barzdins, K.Cerans, R.Liepins, A.Sprogis: OWLGrEd: a UML Style Graphical Editor for OWL, to appear in Proceedings of ORES 2010, ESWC 2010 Workshop on Ontology Repositories and Editors for the Semantic Web, 2010.
14. OWLGrEd, http://owlgred.lumii.lv/
15. Ontology Definition Metamodel. OMG Adopted Specification. Document Number: ptc/2007-09-09, November 2007. http://www.omg.org/docs/ptc/07-09-09.pdf
16. G.Barzdins, S.Rikacovs, M.Zviedris: Graphical Query Language as SPARQL Frontend. In Grundspenkis, J., Kirikova, M., Manolopoulos, Y., Morzy, T., Novickis, L., Vossen, G. (Eds.), Local Proceedings of 13th East-European Conference (ADBIS 2009), pp. 93–107. Riga Technical University, Riga, 2009.
17. Open Government Directive of December 8, 2009: http://www.whitehouse.gov/sites/default/files/omb/assets/memoranda_2010/m10-06.pdf
18. The UK public data website, http://data.gov.uk
19. TED2009 conference, URL: http://conferences.ted.com/TED2009/
20. Christian Perez de Laborda and Stefan Conrad: Bringing Relational Data into the Semantic Web using SPARQL and Relational.OWL Semantic Web and Databases. In Third International Workshop, SWDB 2006, Co-located with ICDE, Atlanta, USA, April 2006
21. Christian Perez de Laborda and Stefan Conrad: Database to Semantic Web Mapping using RDF Query Languages LNCS 4215, pp. 241-254. Springer, Heidelberg, 2006.
22. J.Barrasa, O.Corcho, G.Shen, A.Gomez-Perez: R2O: An extensible and semantically based database-to-ontology mapping language. In: SWDB'04, 2nd Workshop on Semantic Web and Databases, 2004.

23. D2RQ Platform. Treating Non-RDF Relational Databases as Virtual RDF Graphs. http://www4.wiwiss.fu-berlin.de/bizer/D2RQ/spec/
24. C.Blakeley: “RDF Views of SQL Data (Declarative SQL Schema to RDF Mapping)”, OpenLink Software, 2007.
25. Wu, Z., Chen, H., Wang, H., Wang, Y., Mao, Y., Tang, J., Zhou, C.: “Dartgrid: a Semantic Web Toolkit for Integrating Heterogeneous Relational Databases”, Semantic Web Challenge at 4th International Semantic Web Conference (ISWC 2006), Athens, USA, 5-9 November 2006.
26. Sequeda, J.F., Cunningham, C., Depena, R., Miranker, D.P. Ultrawrap: Using SQL Views for RDB2RDF. In Poster Proceedings of the 8th International Semantic Web Conference (ISWC2009), Chantilly, VA, USA. (2009)
27. Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., Aumueller, D.: Triplify: Light-weight linked data publication from relational databases. In Proceedings of the 18th International Conference on World Wide Web (2009).
28. W3C RDB2RDF Working Group, http://www.w3.org/2001/sw/rdb2rdf/
29. A Survey of Current Approaches for Mapping of Relational Databases to RDF, http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf
30. A Direct Mapping of Relational Data to RDF, http://www.w3.org/TR/rdb-direct-mapping/
31. R2RML: RDB to RDF Mapping Language, http://www.w3.org/TR/r2rml/
32. Turtle - Terse RDF Triple Language, http://www.w3.org/TeamSubmission/turtle/
33. Bizer, C., Schultz, A.: The R2R Framework: Publishing and Discovering Mappings on the Web. 1st International Workshop on Consuming Linked Data (COLD 2010), Shanghai, November 2010.
34. Spyder tool, URL: http://www.revelytix.com/content/spyder
35. Codd, E.F. (1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM 13 (6): 377–387. doi:10.1145/362384.362685.
36. S.Rikacovs, J.Barzdins, Export of Relational Databases to RDF Databases: a Case Study, in P. Forbrig and H. Günther (eds.), Perspectives in Business Informatics Research, Springer LNBIP 64 (2010), 203-211.
37. S.Rikacovs, Export of Relational Databases to RDF Databases by Model Transformation, in Proc. of BIR 2011, Riga, Latvia, October 7-8, 2010. LNBIP 90, pp. 142-157. Springer, Heidelberg, 2011 (ISBN:978-3-642-24510-7)
38. G.Barzdins, J.Barzdins, K.Cerans: From Databases to Ontologies, Semantic Web Engineering in the Knowledge Society; J.Cardoso, M.Lytras (Eds.), IGI Global, 2008 (ISBN: 978-1-60566-112-4) pp. 242-266
39. T.Berners-Lee: Relational Databases on the Semantic Web. http://www.w3.org/DesignIssues/RDB-RDF.html, 1998.
40. Chen H., Wang Y., Wang H., Mao Y., Tang J., Zhou C., Yin A., Wu Z.:Towards a Semantic Web of Relational Databases: a Practical Semantic Toolkit and an In-Use Case from Traditional Chinese Medicine
41. Semantic SQL: http://semanticsql.com/
42. OMG's MetaObject Facility, http://www.omg.org/mof/
43. MOF QVT, http://www.omg.org/spec/QVT/1.0/
44. MOLA resources. URL:http://mola.mii.lu.lv/
45. The <AGG> Homepage, http://user.cs.tu-berlin.de/~gragra/agg/
46. Data for Adam and Eve's Posterity. http://www.johnpratt.com/items/docs/adam_gen/adam.html#
47. Relational.OWL application http://sourceforge.net/projects/relational-owl/
48. RDQuery application http://sourceforge.net/projects/rdquery/
49. Relational.OWL ontology http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#

50. Relational.OWL Application documentation http://dbs.cs.uni-duesseldorf.de/RDF/docs/ROWLApp/
51. SPARQL 1.1 Query Language, http://www.w3.org/TR/sparql11-query/
52. SPARQL New Features and Rationale. W3C Working Draft 2 July 2009 http://www.w3.org/TR/sparql-features/
53. Konstantinos Makris, Nikos Bikakis, Nektarios Gioldasis, Chrisa Tsinaraki, Stavros Christodoulakis: Towards a Mediator Based on OWL and SPARQL. In WSKS 2009, 2nd World Summit on the Knowledge Society, Lecture Notes In Artificial Intelligence; Vol. 5736, pp. 326 - 335. Springer-Verlag, Berlin, Heidelberg, 2009
54. Christian Bizer, Richard Cyganiak: D2RQ — Lessons Learned. Position paper for the W3C Workshop on RDF Access to Relational Databases, 2007. url: http://www.w3.org/2007/03/RdfRDB/papers/d2rq-positionpaper/
55. Jena- A Semantic Web Framework for Java. http://jena.sourceforge.net/
56. Sesame- Java based framework for storage, inferencing and querying of RDF data. http://www.openrdf.org/
57. ODEMapster engine. url: http://neon-toolkit.org/wiki/ODEMapster
58. NeOn toolkit. url: http://neon-toolkit.org
59. N. Cullot, R. Ghawi and K. Yetongnon. In Proc. of 15th Italian Symposium on Advanced Database Systems (SEBD 2007), pages 491-494, Torre Canne, Italy, 17-20 June 2007,url: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.5970&rep=rep1&type=pdf
60. R. Ghawi, N. Cullot: Database-to-Ontology Mapping Generation for Semantic Interoperability. A slideshow, http://www.slideshare.net/rajighawi/db2owl
61. Triplify website, URL: http://Triplify.org.
62. Spyder tool, URL: http://www.revelytix.com/content/spyder
63. Revelytix RDB Mapping Language Specification, http://www.knoodl.com/ui/groups/Mapping_Ontology_Community/file/63791?name=RDB_Mapping_Specification_v0.2
64. R.Fagin, L. Haas, M. Hernandez, R. Miller, L. Popa, Y. Velegrakis: Clio: Schema Mapping Creation and Data Exchange. In Conceptual Modeling: Foundations and Applications, 2009.
65. G.Bumans, Mapping between Relational Databases and OWL Ontologies: an Example, Scientific Papers of University of Latvia, Computer Science and Information Technologies, vol.756, 2010., http://www.lu.lv/materiali/apgads/raksti/756_pp_99-117.pdf
66. Y.An, A.Borgida, J.Mylopoulos: Inferring complex semantic mappings between relational tables and ontologies from simple correspondences. In: OTM'05, On The Move Federated Conference, 2005.
67. Barrasa,J., Gómez-Pérez, A, Upgrading relational legacy data to the semantic web, In Proc. of 15th international conference on World Wide Web Conference (WWW 2006), pages 1069-1070, Edinburgh, United Kingdom, 23-26 May 2006.
68. OpenLink Virtuoso Platform. Automated Generation of RDF Views over Relational Data Sources. URL: http://docs.openlinksw.com/virtuoso/rdfrdfviewgnr.html
69. Object Management Group MOF QVT Final Adopted Specification. URL: http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf
70. ATLAS Model Transformation Language. URL:http://www.eclipse.org/m2m/atl/
71. Eclipse Modeling Framework Project (EMF). URL: http://www.eclipse.org/modeling/emf/
72. Semantic SQL: http://semanticsql.com/
73. Linked data: http://linkeddata.org/

74. ANTLRWorks: The ANTLR GUI Development Environment, URL: http://www.antlr.org/works/index.html
75. Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator, URL: http://javacc.java.net/
76. J.Moden. The "Numbers" or "Tally" Table: What it is and how it replaces a loop. http://www.sqlservercentral.com/articles/T-SQL/62867/
77. Matthew Horridge, Sean Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. OWLED 2009, 6th OWL Experienced and Directions Workshop, Chantilly, Virginia, October 2009.URL: http://www.webont.org/owled/2009/papers/owled2009_submission_29.pdf
78. Renars Liepins, lQuery: A Model Query and Transformation Library, Scientific Papers of University of Latvia, Computer Science and Information Technologies, vol.770, 2010. http://www.lu.lv/apgads/izdevumi/lu-raksti-pdf/770-sejums/
79. Lua, the programming language, URL: http://www.lua.org/

Apendices

A.1 Relational.OWL platform

A.1.1 DDL SQL transformation patterns to Relational.OWL instances

Data Definition Language (DDL) SQL statement transformation to OWL described in this section is not taken from original contribution of Cristian P´erez de Laborda, Stefan Conrad but are deduced from what they described in papers [20, 21]. If definition of DB schema is given as a list of SQL statements then automatic process of creating *Relational.OWL* instance is possible based on given below translation patterns.

Pattern 1.

SQL command for table definition in the folowing pattern, where *tab, col(1), …col(n), type(1), …type(n)* and *comment* are variables and n- natural number

```
CREATE TABLE tab
(
  col(1) db_type(1) PRIMARY KEY,
  col(2) db_type(2),
…
  col(n) db_type(n),
);
COMMENT ON TABLE tab is comment;
```

is translated to the folowing OWL class definition code to represent DB table

```
<rdf:RDF xmlns="http://lumii.lv/mini_university_schema#"
   xmlns:dbs="http://www.dbs.cs.uni-
duesseldorf.de/RDF/relational.owl#"
...
>
...
<owl:Class rdf:ID="tab">
   <rdf:type rdf:resource="&dbs;Table"/>
   <rdfs:label>comment</rdfs:label>
   <dbs:hasColumn rdf:resource="#tab.col(1)"/>
   <dbs:hasColumn rdf:resource="#tab.col(2)"/>
…
   <dbs:hasColumn rdf:resource="#tab.col(n)"/>
   <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
         <dbs:hasColumn rdf:resource="#tab.col(1)"/>
      </dbs:PrimaryKey>
   </dbs:isIdentifiedBy>
</owl:Class>
```

Pattern 2.

SQL command for table definition in the following pattern, where *tab, col(1), …col(n), type(1), …type(n)* and *comment* are variables, n- natural number and p1, …pm- natural numbers from set {1, 2, …n}

```
CREATE TABLE tab
(
  Col(1) db_type(1),
  col(2) db_type(2),
…
  col(n) db_type(n),
  PRIMARY KEY (col(p1), col(p2), … , col(pm) )
);
COMMENT ON TABLE tab is comment;
```

is translated to the folowing OWL class definition code to represent DB table

```
<owl:Class rdf:ID="tab">
   <rdf:type rdf:resource="&dbs;Table"/>
   <rdfs:label>comment</rdfs:label>
   <dbs:hasColumn rdf:resource="#tab.col(1)"/>
   <dbs:hasColumn rdf:resource="#tab.col(2)"/>

…
   <dbs:hasColumn rdf:resource="#tab.col(n)"/>
   <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
         <dbs:hasColumn rdf:resource="#tab.col(p1)"/>
         <dbs:hasColumn rdf:resource="#tab.col(p2)"/>
…
         <dbs:hasColumn rdf:resource="#tab.col(pm)"/>
      </dbs:PrimaryKey>
   </dbs:isIdentifiedBy>
</owl:Class>
```

Pattern 3.

SQL command for table definition in the folowing pattern, where *col(1), ...col(n), type(1), ...type(n)* and *comment* are variables and n- natural number

```
CREATE TABLE tab
(
  Col(1) db_type(1),
  col(2) db_type(2),
…
  col(n) db_type(n),
);
COMMENT ON COLUMN tab.col(1) is comment(1);
COMMENT ON COLUMN tab.col(2) is comment(2);
…
COMMENT ON COLUMN tab.col(n) is comment(n);
```

is translated to the folowing OWL class definition codes for DB table column definition. For each i from 1 to n:

```
<owl:DatatypeProperty rdf:ID="tab.col(i)">
   <rdfs:label> comment(i) </rdfs:label>
   <rdf:type rdf:resource="&dbs;Column"/>
   <rdfs:domain rdf:resource="#tab"/>
   <rdfs:range rdf:resource="&xsd;
        xsd_type_for(type(i))" />
</owl:DatatypeProperty>
```

xsd_type_for(type(i)) denotes xsd type corresponding to the type of DB table column. Some correspondences are lested below.

SQL type	XSD type
CHAR(n)	&xsd;string
VARCHAR(n)	&xsd;string
NUMBER(n,m)	&xsd;decimal, specifying totalDigits and fractionDigits
INTEGER	&xsd;integer
INTEGER ar ierobežojumu >0	xsd;positiveInteger
DATE	&xsd;date
DATETIME	&xsd;datetime
BOOLEAN	&xsd;Boolean
…	…

Pattern 4.

SQL statement for foreign key creation with the folowing pattern where *tab(1), tab(2), col(1), col(2)* and *FK_name* are variables

```
ALTER TABLE tab(1)
   ADD CONSTRAINT FK_name FOREIGN KEY (col(1))
      references tab(2) (col(2));
```

is translated to the folowing OWL DatatypeProperty constraint „dbs:references" according to algorithm: first the OWL code is found that describes tab.col(i) column definition according to Pattern 3 and then „<dbs:references …" is added before closing </owl.DatatypeProperty>:

```
<owl:DatatypeProperty rdf:ID="tab(1).col(1)">
…
   <dbs:references rdf:resource="#tab(2).col(2)"/>
</owl:DatatypeProperty>
```

A.1.2 RDB schema transformation to OWL

This section describes transformation from relational schema to OWL ontology ROWL that is instance of *Relational.OWL* ontology. Relational schema from mini-university example [2.3.1] will be used. Table and column description will be according to pattern described in section [A.1.1].

First thing to describe is relational database schema and tables belonging to it. Namespace dbs to Relationa.OWL ontology is defined also.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
  <!ENTITY dbs "http://www.dbs.cs.uni-
duesseldorf.de/RDF/relational.owl#">
]>
<rdf:RDF xmlns="http://lumii.lv/mini_university_schema#"
  xmlns:dbs=http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#
  xml:base="http://lumii.lv/mini_university_schema#"
…
>
<owl:Ontology rdf:about="mini_university_rowl"></owl:Ontology>
<owl:Class rdf:ID="MINI_UNIVERSITY">
```

```
    <rdf:type rdf:resource="&dbs;Database"/>
    <dbs:hasTable rdf:resource="#COURSE"/>
    <dbs:hasTable rdf:resource="#STUDENT"/>
    <dbs:hasTable rdf:resource="#REGISTRATION"/>
    <dbs:hasTable rdf:resource="#TEACHER"/>
    <dbs:hasTable rdf:resource="#TEACHER_LEVEL"/>
    <dbs:hasTable rdf:resource="#PROGRAM"/>
  </owl:Class>
```

Here 6 classes for table definitions are referenced. We show one of them. Class for table *course* is described listing all tables by means of OWL datatype property *dbs:hasColumn*. Here class *COURSE* is defined as being instance of class *Table* from *Relational.OWL* ontology. Class being instance of another class means that OWL Full language is used.

```
  <owl:Class rdf:ID="COURSE">
    <rdf:type rdf:resource="&dbs;Table"/>
    <dbs:hasColumn rdf:resource="#COURSE.COURSE_ID"/>
    <dbs:hasColumn rdf:resource="#COURSE.TEACHER_ID"/>
    <dbs:hasColumn rdf:resource="#COURSE.PROGRAM_ID"/>
    <dbs:hasColumn rdf:resource="#COURSE.NAME"/>
    <dbs:hasColumn rdf:resource="#COURSE.REQUIRED"/>
    <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
        <dbs:hasColumn rdf:resource="#COURSE.COURSE_ID"/>
      </dbs:PrimaryKey>
    </dbs:isIdentifiedBy>
  </owl:Class>
```

DB table columns ar described by OWL datatype properties that in the same time are instances of *Column* class from *Relational.OWL* ontology. This again requires OWL Full usage. Columns references to other columns (FK keys) are recorded by *dbs:references* property.

```
    <owl:DatatypeProperty rdf:ID="COURSE.COURSE_ID">
      <rdf:type rdf:resource="&dbs;Column"/>
      <rdfs:domain rdf:resource="#COURSE"/>
      <rdfs:range rdf:resource="&xsd;int"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:ID="COURSE.TEACHER_ID">
      <rdf:type rdf:resource="&dbs;Column"/>
      <rdfs:domain rdf:resource="#COURSE"/>
      <rdfs:range rdf:resource="&xsd;int"/>
      <dbs:references rdf:resource="#TEACHER.TEACHER_ID"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:ID="COURSE.PROGRAM_ID">
      <rdf:type rdf:resource="&dbs;Column"/>
      <rdfs:domain rdf:resource="#COURSE"/>
      <rdfs:range rdf:resource="&xsd;int"/>
      <dbs:references rdf:resource="#PROGRAM.PROGRAM_ID"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:ID="COURSE.NAME">
      <rdf:type rdf:resource="&dbs;Column"/>
      <rdfs:domain rdf:resource="#PERSON"/>
      <rdfs:range rdf:resource="&xsd;string"/>
      <dbs:length>40</dbs:length>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:ID="COURSE.REQUIRED">
      <rdf:type rdf:resource="&dbs;Column"/>
```

```
    <rdfs:domain rdf:resource="#PERSON"/>
    <rdfs:range rdf:resource="&xsd;int"/>
    <dbs:length>1</dbs:length>
  </owl:DatatypeProperty>
```

Full source code for ROWL for mini-university example is given in apendice [A.1.1]

A.1.3 RDB data transformation to RDF

To get RDF triple set acording relational data we need to create instances of clases and properties in ROWL ontology (described in previous section [A.1.1]). Classes for tables are named (rdf:ID) as pattern TABLE_NAME, datatype property ar named as pattern TABLE_NAME.COLUMN_NAME. The triple set for one row for table TABLE_NAME that has n columns is obtained in folowing pattern:

```
<x> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> TABLE_NAME .
<x> <URI_OF_ROWL#TABLE_NAME.COLUMN_NAME_1> COLUMN_VALUE_1 .
<x> <URI_OF_ROWL#TABLE_NAME.COLUMN_NAME_2> COLUMN_VALUE_2 .
…
<x> <URI_OF_ROWL#TABLE_NAME.COLUMN_NAME_n> COLUMN_VALUE_n .
```

Here we are not assigning any URI to subjects as one of columns should be primary key so any blank note for x sufices. For other rows and tables different blank notes should be used. Taking these requirenments into account the above given triples can be rewriten in RDF/XML notion:

```
<rdf:RDF
  xmlns="URI_OF_ROWL"
>
<TABLE_NAME>
  <TABLE_NAME.COLUMN_NAME_1>
    COLUMN_VALUE_1
  </TABLE_NAME.COLUMN_NAME_1>
</TABLE_NAME>
<TABLE_NAME>
  <TABLE_NAME.COLUMN_NAME_2>
    COLUMN_VALUE_2
  </TABLE_NAME.COLUMN_NAME_2>
</TABLE_NAME>
…
<TABLE_NAME>
  <TABLE_NAME.COLUMN_NAME_n>
    COLUMN_VALUE_n
  </TABLE_NAME.COLUMN_NAME_n>
</TABLE_NAME>
```

Some of RDF triples for mini-university example data (described in section [2.3.1]) are, assuming URI_OF_ROWL=http://lumii.lv/mini_university_schema#:

```
<rdf:RDF
  xmlns="http://lumii.lv/mini_university_schema#"
  xml:base="http://lumii.lv/mini_university_data#"
>
<PROGRAM>
  <PROGRAM.PROGRAM_ID>1</PROGRAM.PROGRAM_ID>
  <PROGRAM.NAME>Computer Science</PROGRAM.NAME>
</PROGRAM>
```

```
<PROGRAM>
  <PROGRAM.PROGRAM_ID>2</PROGRAM.PROGRAM_ID>
  <PROGRAM.NAME>Computer Engeneering</PROGRAM.NAME>
</PROGRAM>

<TEACHER_LEVEL>
  <TEACHER_LEVEL.LEVEL_CODE>Assistant</TEACHER_LEVEL.LEVEL_CODE>
</TEACHER_LEVEL>
<TEACHER_LEVEL>
  <TEACHER_LEVEL.LEVEL_CODE>Associate
Professor</TEACHER_LEVEL.LEVEL_CODE>
</TEACHER_LEVEL>
<TEACHER_LEVEL>
  <TEACHER_LEVEL.LEVEL_CODE>Professor</TEACHER_LEVEL.LEVEL_CODE>
</TEACHER_LEVEL>
<TEACHER>
  <TEACHER.TEACHER_ID>1</TEACHER.TEACHER_ID>
  <TEACHER.LEVEL_CODE>Professor</TEACHER.LEVEL_CODE>
  <TEACHER.IDCODE>999999999</TEACHER.IDCODE>
  <TEACHER.NAME>Alice</TEACHER.NAME>
</TEACHER>
<TEACHER>
  <TEACHER.TEACHER_ID>2</TEACHER.TEACHER_ID>
  <TEACHER.LEVEL_CODE>Professor</TEACHER.LEVEL_CODE>
  <TEACHER.IDCODE>777777777</TEACHER.IDCODE>
  <TEACHER.NAME>Bob</TEACHER.NAME>
</TEACHER>
…
```

Full source code for RDF triples for mini-university example is given in apendice [A.1.5]

A.1.4 Relational.OWL ontology for mini-university example database schema

mini_university_schema.owl code

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
  <!ENTITY dbs "http://www.dbs.cs.uni-
duesseldorf.de/RDF/relational.owl#">
]>
<rdf:RDF xmlns="http://lumii.lv/mini_university_schema#"
  xmlns:dbs="http://www.dbs.cs.uni-
duesseldorf.de/RDF/relational.owl#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://lumii.lv/mini_university_schema#"
>
<owl:Ontology rdf:about=""></owl:Ontology>
<!--
DB Schema for mini_university
-->
  <owl:Class rdf:ID="MINI_UNIVERSITY">
    <rdf:type rdf:resource="&dbs;Database"/>
    <dbs:hasTable rdf:resource="#COURSE"/>
    <dbs:hasTable rdf:resource="#STUDENT"/>
    <dbs:hasTable rdf:resource="#REGISTRATION"/>
```

```
    <dbs:hasTable rdf:resource="#TEACHER"/>
    <dbs:hasTable rdf:resource="#TEACHER_LEVEL"/>
    <dbs:hasTable rdf:resource="#PROGRAM"/>
  </owl:Class>
<!--
  Table COURSE
-->
  <owl:Class rdf:ID="COURSE">
    <rdf:type rdf:resource="&dbs;Table"/>
    <dbs:hasColumn rdf:resource="#COURSE.COURSE_ID"/>
    <dbs:hasColumn rdf:resource="#COURSE.TEACHER_ID"/>
    <dbs:hasColumn rdf:resource="#COURSE.PROGRAM_ID"/>
    <dbs:hasColumn rdf:resource="#COURSE.NAME"/>
    <dbs:hasColumn rdf:resource="#COURSE.REQUIRED"/>
    <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
        <dbs:hasColumn rdf:resource="#COURSE.COURSE_ID"/>
      </dbs:PrimaryKey>
    </dbs:isIdentifiedBy>
  </owl:Class>
<!--
  Columns of Table COURSE
-->
  <owl:DatatypeProperty rdf:ID="COURSE.COURSE_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#COURSE"/>
    <rdfs:range rdf:resource="&xsd;int"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="COURSE.TEACHER_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#COURSE"/>
    <rdfs:range rdf:resource="&xsd;int"/>
    <dbs:references rdf:resource="#TEACHER.TEACHER_ID"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="COURSE.PROGRAM_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#COURSE"/>
    <rdfs:range rdf:resource="&xsd;int"/>
    <dbs:references rdf:resource="#PROGRAM.PROGRAM_ID"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="COURSE.NAME">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#PERSON"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>40</dbs:length>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="COURSE.REQUIRED">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#PERSON"/>
    <rdfs:range rdf:resource="&xsd;int"/>
    <dbs:length>1</dbs:length>
  </owl:DatatypeProperty>

<!--
  Table STUDENT
-->
  <owl:Class rdf:ID="STUDENT">
    <rdf:type rdf:resource="&dbs;Table"/>
    <dbs:hasColumn rdf:resource="#STUDENT.STUDENT_ID"/>
    <dbs:hasColumn rdf:resource="#STUDENT.PROGRAM_ID"/>
```

```
    <dbs:hasColumn rdf:resource="#STUDENT.IDCODE"/>
    <dbs:hasColumn rdf:resource="#STUDENT.NAME"/>
    <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
        <dbs:hasColumn rdf:resource="#STUDENT.STUDENT_ID"/>
      </dbs:PrimaryKey>
    </dbs:isIdentifiedBy>
  </owl:Class>
<!--
  Columns of Table STUDENT
-->
  <owl:DatatypeProperty rdf:ID="STUDENT.STUDENT_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#STUDENT"/>
    <rdfs:range rdf:resource="&xsd;int"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="STUDENT.PROGRAM_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#STUDENT"/>
    <rdfs:range rdf:resource="&xsd;int"/>
    <dbs:references rdf:resource="#PROGRAM.PROGRAM_ID"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="STUDENT.IDCODE">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#STUDENT"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>30</dbs:length>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="STUDENT.NAME">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#STUDENT"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>80</dbs:length>
  </owl:DatatypeProperty>

<!--
  Table REGISTRATION
-->
  <owl:Class rdf:ID="REGISTRATION">
    <rdf:type rdf:resource="&dbs;Table"/>
    <dbs:hasColumn rdf:resource="#REGISTRATION.REGISTRATION_ID"/>
    <dbs:hasColumn rdf:resource="#REGISTRATION.COURSE_ID"/>
    <dbs:hasColumn rdf:resource="#REGISTRATION.STUDENT_ID"/>
    <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
        <dbs:hasColumn
rdf:resource="#REGISTRATION.REGISTRATION_ID"/>
      </dbs:PrimaryKey>
    </dbs:isIdentifiedBy>
  </owl:Class>
<!--
  Columns of Table REGISTRATION
-->
  <owl:DatatypeProperty rdf:ID="REGISTRATION.REGISTRATION_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#REGISTRATION"/>
    <rdfs:range rdf:resource="&xsd;int"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="REGISTRATION.COURSE_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
```

```
    <rdfs:domain rdf:resource="#REGISTRATION"/>
    <rdfs:range rdf:resource="&xsd;int"/>
    <dbs:references rdf:resource="#COURSE.COURSE_ID"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="REGISTRATION.STUDENT_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#REGISTRATION"/>
    <rdfs:range rdf:resource="&xsd;int"/>
    <dbs:references rdf:resource="#STUDENT.STUDENT_ID"/>
  </owl:DatatypeProperty>

<!--
  Table TEACHER
-->
  <owl:Class rdf:ID="TEACHER">
    <rdf:type rdf:resource="&dbs;Table"/>
    <dbs:hasColumn rdf:resource="#TEACHER.TEACHER_ID"/>
    <dbs:hasColumn rdf:resource="#TEACHER.LEVEL_CODE"/>
    <dbs:hasColumn rdf:resource="#TEACHER.IDCODE"/>
    <dbs:hasColumn rdf:resource="#TEACHER.NAME"/>
    <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
        <dbs:hasColumn rdf:resource="#TEACHER.TEACHER_ID"/>
      </dbs:PrimaryKey>
    </dbs:isIdentifiedBy>
  </owl:Class>
<!--
  Columns of Table TEACHER
-->
  <owl:DatatypeProperty rdf:ID="TEACHER.TEACHER_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#TEACHER"/>
    <rdfs:range rdf:resource="&xsd;int"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="TEACHER.LEVEL_CODE">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#TEACHER"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>30</dbs:length>
    <dbs:references rdf:resource="#TEACHER_LEVEL.LEVEL_CODE"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="TEACHER.IDCODE">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#TEACHER"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>30</dbs:length>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="TEACHER.NAME">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#TEACHER"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>40</dbs:length>
  </owl:DatatypeProperty>

<!--
  Table TEACHER_LEVEL
-->
  <owl:Class rdf:ID="TEACHER_LEVEL">
    <rdf:type rdf:resource="&dbs;Table"/>
    <dbs:hasColumn rdf:resource="#TEACHER_LEVEL.LEVEL_CODE"/>
```

```
    <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
        <dbs:hasColumn rdf:resource="#TEACHER_LEVEL.LEVEL_CODE"/>
      </dbs:PrimaryKey>
    </dbs:isIdentifiedBy>
  </owl:Class>
<!--
  Columns of Table TEACHER_LEVEL
-->
  <owl:DatatypeProperty rdf:ID="TEACHER_LEVEL.LEVEL_CODE">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#TEACHER_LEVEL"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>30</dbs:length>
  </owl:DatatypeProperty>

<!--
  Table PROGRAM
-->
  <owl:Class rdf:ID="PROGRAM">
    <rdf:type rdf:resource="&dbs;Table"/>
    <dbs:hasColumn rdf:resource="#PROGRAM.PROGRAM_ID"/>
    <dbs:hasColumn rdf:resource="#TEACHER.NAME"/>
    <dbs:isIdentifiedBy>
      <dbs:PrimaryKey>
        <dbs:hasColumn rdf:resource="#PROGRAM.PROGRAM_ID"/>
      </dbs:PrimaryKey>
    </dbs:isIdentifiedBy>
  </owl:Class>
<!--
  Columns of Table PROGRAM
-->
  <owl:DatatypeProperty rdf:ID="PROGRAM.PROGRAM_ID">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#PROGRAM"/>
    <rdfs:range rdf:resource="&xsd;int"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="PROGRAM.NAME">
    <rdf:type rdf:resource="&dbs;Column"/>
    <rdfs:domain rdf:resource="#PROGRAM"/>
    <rdfs:range rdf:resource="&xsd;string"/>
    <dbs:length>80</dbs:length>
  </owl:DatatypeProperty>

</rdf:RDF>
```

A.1.5 Relational.OWL ontology instance data for mini-university example

mini_university_data.rdf code

```
<rdf:RDF
  xmlns="http://lumii.lv/mini_university_schema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://lumii.lv/mini_university_data#"
>
<PROGRAM>
```

```
  <PROGRAM.PROGRAM_ID>1</PROGRAM.PROGRAM_ID>
  <PROGRAM.NAME>Computer Science</PROGRAM.NAME>
</PROGRAM>
<PROGRAM>
  <PROGRAM.PROGRAM_ID>2</PROGRAM.PROGRAM_ID>
  <PROGRAM.NAME>Computer Engeneering</PROGRAM.NAME>
</PROGRAM>

<TEACHER_LEVEL>
  <TEACHER_LEVEL.LEVEL_CODE>Assistant</TEACHER_LEVEL.LEVEL_CODE>
</TEACHER_LEVEL>
<TEACHER_LEVEL>
  <TEACHER_LEVEL.LEVEL_CODE>
    AssociateProfessor
  </TEACHER_LEVEL.LEVEL_CODE>
</TEACHER_LEVEL>
<TEACHER_LEVEL>
  <TEACHER_LEVEL.LEVEL_CODE>Professor</TEACHER_LEVEL.LEVEL_CODE>
</TEACHER_LEVEL>

<STUDENT>
  <STUDENT.STUDENT_ID>1</STUDENT.STUDENT_ID>
  <STUDENT.PROGRAM_ID>1</STUDENT.PROGRAM_ID>
  <STUDENT.IDCODE>123456789</STUDENT.IDCODE>
  <STUDENT.NAME>Dave</STUDENT.NAME>
</STUDENT>
<STUDENT>
  <STUDENT.STUDENT_ID>2</STUDENT.STUDENT_ID>
  <STUDENT.PROGRAM_ID>2</STUDENT.PROGRAM_ID>
  <STUDENT.IDCODE>987654321</STUDENT.IDCODE>
  <STUDENT.NAME>Eve</STUDENT.NAME>
</STUDENT>
<STUDENT>
  <STUDENT.STUDENT_ID>3</STUDENT.STUDENT_ID>
  <STUDENT.PROGRAM_ID>1</STUDENT.PROGRAM_ID>
  <STUDENT.IDCODE>555555555</STUDENT.IDCODE>
  <STUDENT.NAME>Charlie</STUDENT.NAME>
</STUDENT>
<STUDENT>
  <STUDENT.STUDENT_ID>4</STUDENT.STUDENT_ID>
  <STUDENT.PROGRAM_ID>2</STUDENT.PROGRAM_ID>
  <STUDENT.IDCODE>345453432</STUDENT.IDCODE>
  <STUDENT.NAME>Ivan</STUDENT.NAME>
</STUDENT>

<TEACHER>
  <TEACHER.TEACHER_ID>1</TEACHER.TEACHER_ID>
  <TEACHER.LEVEL_CODE>Professor</TEACHER.LEVEL_CODE>
  <TEACHER.IDCODE>999999999</TEACHER.IDCODE>
  <TEACHER.NAME>Alice</TEACHER.NAME>
</TEACHER>
<TEACHER>
  <TEACHER.TEACHER_ID>2</TEACHER.TEACHER_ID>
  <TEACHER.LEVEL_CODE>Professor</TEACHER.LEVEL_CODE>
  <TEACHER.IDCODE>777777777</TEACHER.IDCODE>
  <TEACHER.NAME>Bob</TEACHER.NAME>
</TEACHER>
<TEACHER>
  <TEACHER.TEACHER_ID>2</TEACHER.TEACHER_ID>
  <TEACHER.LEVEL_CODE>Assistant</TEACHER.LEVEL_CODE>
```

```
  <TEACHER.IDCODE>555555555</TEACHER.IDCODE>
  <TEACHER.NAME>Charlie</TEACHER.NAME>
</TEACHER>

<COURSE>
  <COURSE.COURSE_ID>1</COURSE.COURSE_ID>
  <COURSE.PROGRAM_ID>2</COURSE.PROGRAM_ID>
  <COURSE.TEACHER_ID>3</COURSE.TEACHER_ID>
  <COURSE.NAME>Programming Basics</COURSE.NAME>
  <COURSE.REQUIRED>0</COURSE.REQUIRED>
</COURSE>
<COURSE>
  <COURSE.COURSE_ID>2</COURSE.COURSE_ID>
  <COURSE.PROGRAM_ID>1</COURSE.PROGRAM_ID>
  <COURSE.TEACHER_ID>1</COURSE.TEACHER_ID>
  <COURSE.NAME>Semantic Web</COURSE.NAME>
  <COURSE.REQUIRED>1</COURSE.REQUIRED>
</COURSE>
<COURSE>
  <COURSE.COURSE_ID>3</COURSE.COURSE_ID>
  <COURSE.PROGRAM_ID>2</COURSE.PROGRAM_ID>
  <COURSE.TEACHER_ID>2</COURSE.TEACHER_ID>
  <COURSE.NAME>Computer Networks</COURSE.NAME>
  <COURSE.REQUIRED>1</COURSE.REQUIRED>
</COURSE>
<COURSE>
  <COURSE.COURSE_ID>4</COURSE.COURSE_ID>
  <COURSE.PROGRAM_ID>1</COURSE.PROGRAM_ID>
  <COURSE.TEACHER_ID>2</COURSE.TEACHER_ID>
  <COURSE.NAME>Quantum Computations</COURSE.NAME>
  <COURSE.REQUIRED>0</COURSE.REQUIRED>
</COURSE>

<REGISTRATION>
  <REGISTRATION.REGISTRATION_ID>1</REGISTRATION.REGISTRATION_ID>
  <REGISTRATION.STUDENT_ID>1</REGISTRATION.STUDENT_ID>
  <REGISTRATION.COURSE_ID>2</REGISTRATION.COURSE_ID>
</REGISTRATION>
<REGISTRATION>
  <REGISTRATION.REGISTRATION_ID>2</REGISTRATION.REGISTRATION_ID>
  <REGISTRATION.STUDENT_ID>1</REGISTRATION.STUDENT_ID>
  <REGISTRATION.COURSE_ID>4</REGISTRATION.COURSE_ID>
</REGISTRATION>
<REGISTRATION>
  <REGISTRATION.REGISTRATION_ID>3</REGISTRATION.REGISTRATION_ID>
  <REGISTRATION.STUDENT_ID>2</REGISTRATION.STUDENT_ID>
  <REGISTRATION.COURSE_ID>1</REGISTRATION.COURSE_ID>
</REGISTRATION>
<REGISTRATION>
  <REGISTRATION.REGISTRATION_ID>4</REGISTRATION.REGISTRATION_ID>
  <REGISTRATION.STUDENT_ID>2</REGISTRATION.STUDENT_ID>
  <REGISTRATION.COURSE_ID>3</REGISTRATION.COURSE_ID>
</REGISTRATION>
<REGISTRATION>
  <REGISTRATION.REGISTRATION_ID>5</REGISTRATION.REGISTRATION_ID>
  <REGISTRATION.STUDENT_ID>3</REGISTRATION.STUDENT_ID>
  <REGISTRATION.COURSE_ID>2</REGISTRATION.COURSE_ID>
</REGISTRATION>

</rdf:RDF>
```

A.1.6 SPARQL scripts to map ROWL (Relational.OWL instance) to target ontology and listing for mini-university example

SPARQL statement list can implement mapping between not only ROWL and target ontology but between any two separate ontologies. Mappings are implementated as a list of SPARQL statements *map_1*, *map_2*, …, *map_n* each in form, defining triples for target ontology in *CONSTRUCT* clause and selecting triples from source ontology(ies) in WHERE clause:

```
CONSTRUCT
{
  target_triple_patterns
}
WHERE
{
  sourse_tripple_patterns
}
```

To get all target ontology triples one need to execute all the mapping construct queries *map_1*, *map_2*, …, *map_n*, merging obtained triple sets.

The mappings by SPARQL will be illustrated for mini-iniversity example [2.3.1]. In examples *schema* namespace is for ROWL ontology and *target* namespace- for target ontology. Next mapping SPARQL maps *TEACHER* table rows having "Professor" value in *LEVEL_CODE* field to *Professor* class instances in target ontology:

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?teacher a target:Professor
}
WHERE {
  ?teacher a schema:TEACHER ;
           schema:TEACHER.LEVEL_CODE "Professor"
}
```

The next mapping is for OWL object propery *takes* having *Student* class as domain and *Course* class as range. In Database tables STUDENT and COURSE are in n:n relation through third table REGISTRATION. The joins are done in a similar way as it would be done in SQL (prefix definitions omited in following mapping scripts):

```
CONSTRUCT {
  ?student target:takes ?course
}
WHERE {
  ?student a schema:STUDENT ;
           schema:STUDENT.STUDENT_ID ?studentId .
  ?registration schema:REGISTRATION.STUDENT_ID ?studentId  ;
           schema:REGISTRATION.COURSE_ID ?courseId  .
  ?course schema:COURSE.COURSE_ID ?courseId  .
}
```

Mapping for *Student* class instances together with data property *personName*:

```
CONSTRUCT {
  ?student a target:Student ;
         target:personName ?studentName
}
WHERE {
  ?student a schema:STUDENT ;
```

```
            schema:STUDENT.NAME ?studentName
}
```

Executing it in triple store where ROWL instances are loaded (tried in Sesame repository) the folowing triples were generated (subject blank nodes simplified)

Table 40. Generated instances for Student class

Subject	Predicate	Object
_:n1	<type>	<target:Student>
_:n1	<target:personName>	"Dave"
_:n2	<type>	<target:Student>
_:n2	<target:personName>	"Eve"
_:n3	<type>	<target:Student>
_:n3	<target:personName>	"Charlie"
_:n4	<type>	<target:Student>
_:n4	<target:personName>	"Ivan"

The most nontrivial mapping is for Class *PersonID* and its property *IDvalue*. Instance data are taken from two tables *STUDENT* and *TEACHER* filling property *Idvalue* from *STUDENT.IDCODE* and *TEACHER.IDCODE* fields and finally creating links from Student and Teacher classe instances to PersonID instances (property *personID*):

```
CONSTRUCT {
  _:x a target:PersonID ;
         target:IDvalue ?idvalue .
  ?person target:personID _:x
}
WHERE
{
  {
    ?person a schema:TEACHER ;
             schema:TEACHER.IDCODE ?idvalue  .
  }
  UNION
  {
    ?person a schema:STUDENT ;
             schema:STUDENT.IDCODE ?idvalue  .
  }
}
```

Here ?person in CONSTRUCT clause creates instance of Student or Teacher class depending from which side of UNION it is filled: if from "?person a schema:TEACHER" then Teacher class otherwise Student class. This is because of other mapping scripts: from the first mapping script shown above: "?person a schema:TEACHER" →*Professor* instance →*Teacher* instance (as superclass).

Mapping script class_AcademicProgram.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?program a target:AcademicProgram ;
         target:programName ?programName
}
WHERE {
```

```
  ?program a schema:PROGRAM ;
           schema:PROGRAM.NAME ?programName
}
```

Mapping script class_Assistant.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?teacher a target:Assistant
}
WHERE {
  ?teacher a schema:TEACHER ;
           schema:TEACHER.LEVEL_CODE "Assistant"
}
```

Mapping script class_AssocProfessor.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?teacher a target:AssocProfessor
}
WHERE {
  ?teacher a schema:TEACHER ;
           schema:TEACHER.LEVEL_CODE "AssocProfessor"
}
```

Mapping script class_Professor.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?teacher a target:Professor
}
WHERE {
  ?teacher a schema:TEACHER ;
           schema:TEACHER.LEVEL_CODE "Professor"
}
```

Mapping script class_Teacher.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?teacher a target:Teacher ;
         target:personName ?teacherName ;
         target:teaches ?course
}
WHERE {
  ?teacher a schema:TEACHER ;
           schema:TEACHER.NAME ?teacherName ;
           schema:TEACHER.TEACHER_ID ?teacherId .
  ?course schema:COURSE.TEACHER_ID  ?teacherId
}
```

Mapping script class_MandatoryCourse.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?course a target:MandatoryCourse ;
         target:courseName ?courseName
}
WHERE {
  ?course a schema:COURSE ;
          schema:COURSE.NAME ?courseName ;
          schema:COURSE.REQUIRED "1"
}
```

Mapping script class_OptionalCourse.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?course a target:OptionalCourse ;
         target:courseName ?courseName
}
WHERE {
  ?course a schema:COURSE ;
          schema:COURSE.NAME ?courseName ;
          schema:COURSE.REQUIRED "0"
}
```

Mapping script class_Student.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?student a target:Student ;
         target:personName ?studentName
}
WHERE {
  ?student a schema:STUDENT ;
           schema:STUDENT.NAME ?studentName .
}
```

Mapping script property_enrolled.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?student target:enrolled ?program
}
WHERE {
  ?student a schema:STUDENT ;
           schema:STUDENT.PROGRAM_ID ?programId .
  ?program a schema:PROGRAM ;
           schema:PROGRAM.PROGRAM_ID ?programId .
}
```

Mapping script property_includes.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?program target:includes ?course
}
WHERE {
  ?program a schema:PROGRAM .
  ?course  a schema:COURSE ;
           schema:COURSE.PROGRAM_ID  ?programId .
  ?program schema:PROGRAM.PROGRAM_ID ?programId
}
```

Mapping script property_takes.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  ?student target:takes ?course
}
WHERE {
  ?student a schema:STUDENT ;
           schema:STUDENT.STUDENT_ID ?studentId .
  ?registration schema:REGISTRATION.STUDENT_ID ?studentId  ;
           schema:REGISTRATION.COURSE_ID ?courseId  .
```

```
  ?course schema:COURSE.COURSE_ID ?courseId  .
}
```

Mapping script class_PersonID.sparql

```
PREFIX schema:<http://lumii.lv/mini_university_schema#>
PREFIX target:<http://lumii.lv/mini_university#>
CONSTRUCT {
  _:x a target:PersonID ;
        target:IDvalue ?idvalue .
  ?person target:personID _:x
}
WHERE
{
  {
    ?person a schema:TEACHER ;
              schema:TEACHER.IDCODE ?idvalue  .
  }
  UNION
  {
    ?person a schema:STUDENT ;
              schema:STUDENT.IDCODE ?idvalue  .
  }
}
```

A.2 D2RQ platform

A.2.1 D2RQ mapping script for mini-university example [2.3.1]

```
@prefix map: <file:/C:/semantic_web/d2r-server-
0.7/school_mapping.n3#> .
@prefix db: <> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d2rq: <http://www.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/0.1#> .
@prefix jdbc: <http://d2rq.org/terms/jdbc/> .
@prefix ex: <http://lumii.lv/ex#> .

map:database a d2rq:Database;
 d2rq:jdbcDriver "oracle.jdbc.driver.OracleDriver";
 d2rq:jdbcDSN "jdbc:oracle:thin:@guntars-PC:1521:gun";
 d2rq:username "school1";
 d2rq:password "s";
 .
# Course class
map:Course a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "course@@XCOURSE.COURSE_ID@@";
 d2rq:class ex:Course;
 .
# property bridge for OptionalCourse
map:OptionalCourse a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Course;
       d2rq:property rdf:type;
       d2rq:condition "required=0";
```

```
       d2rq:constantValue ex:OptionalCourse;
 .
# property bridge for MandatoryCourse class
map:MandatoryCourse a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Course;
       d2rq:property rdf:type;
       d2rq:condition "required=1";
       d2rq:constantValue ex:MandatoryCourse;
 .
# courseName property
 map:courseName a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Course;
 d2rq:property ex:courseName;
 d2rq:column "XCOURSE.NAME";
 d2rq:datatype xsd:string;
 .
#  Teacher class
map:Teacher a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "teacher@@XTEACHER.TEACHER_ID@@";
 d2rq:class ex:Teacher;
 .
# property bridge for Assistant class (subclass of Teacher)
map:Assistant a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Teacher;
       d2rq:property rdf:type;
       d2rq:condition "level_code = 'Assistant'";
       d2rq:constantValue ex:Assistant;
    .
# property bridge for Professor class (subclass of Teacher)
map:Professor a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Teacher;
       d2rq:property rdf:type;
       d2rq:condition "level_code = 'Professor'";
       d2rq:constantValue ex:Professor;
    .
# property bridge for AssocProfessor class (subclass of Teacher)
map:AssocProfessor a d2rq:PropertyBridge;
       d2rq:belongsToClassMap map:Teacher;
       d2rq:property rdf:type;
       d2rq:condition "level_code = 'Associate Professor'";
       d2rq:constantValue ex:AssocProfessor;
    .
# personName property bridges for Teacher class
map:personName_Teacher a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Teacher;
 d2rq:property ex:personName;
 d2rq:column "XTEACHER.NAME";
 d2rq:datatype xsd:string;
 .
# class map for Student class (subclass of Person)
map:Student a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "student@@XSTUDENT.STUDENT_ID@@";
 d2rq:class ex:Student;
 .
# property map for personName for Student domain
map:personName_Student a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Student;
 d2rq:property ex:personName;
```

```
 d2rq:column "XSTUDENT.NAME";
 d2rq:datatype xsd:string;
 .
# class map for AcademicProgram class
map:AcademicProgram a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "program@@XPROGRAM.PROGRAM_ID@@";
 d2rq:class ex:AcademicProgram;
 .
map:programName a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:AcademicProgram;
 d2rq:property ex:programName;
 d2rq:column "XPROGRAM.NAME";
 d2rq:datatype xsd:string;
 .
# 1. class map for PersonID
map:PersonID_teacher a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "personID@@XTEACHER.IDCODE@@";
 d2rq:class ex:PersonID;
 .
map:IDValue1 a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:PersonID_teacher;
 d2rq:property ex:IDValue;
 d2rq:column "XTEACHER.IDCODE";
 d2rq:datatype xsd:string;
 .
# 2. class map for PersonID
map:PersonID_student a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "personID@@XSTUDENT.IDCODE@@";
 d2rq:class ex:PersonID;
 .
map:IDValue2 a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:PersonID_student;
 d2rq:property ex:IDValue;
 d2rq:column "XSTUDENT.IDCODE";
 d2rq:datatype xsd:string;
 .
# Now comes object properties
# object property teaches between Teacher and Course:

map:teaches a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Teacher;
    d2rq:property ex:teaches;
    d2rq:refersToClassMap map:Course;
    d2rq:join "XTEACHER.TEACHER_ID <= XCOURSE.TEACHER_ID";
    .
# object property includes between AcademicProgram and Course
map:includes a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:AcademicProgram;
    d2rq:property ex:includes;
    d2rq:refersToClassMap map:Course;
    d2rq:join "XPROGRAM.PROGRAM_ID => XCOURSE.PROGRAM_ID ";
    .
#object property enrolled Student and AcademicProgram
map:enrollod a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Student;
    d2rq:property ex:enrolled;
    d2rq:refersToClassMap map:AcademicProgram;
```

```
    d2rq:join "XSTUDENT.PROGRAM_ID => XPROGRAM.PROGRAM_ID ";
.
map:takes a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Student;
    d2rq:property ex:takes;
    d2rq:refersToClassMap map:Course;
    d2rq:join "XSTUDENT.STUDENT_ID <= XREGISTRATION.STUDENT_ID ";
    d2rq:join "XREGISTRATION.COURSE_ID => XCOURSE.COURSE_ID";
.
# object property personID between classes Person and PersonID
# one property bridge connects Person class map (1 of 2)
# which is for XStudent table to PersonID class map for Xstudent
# table (a longer version)
map:personID1 a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Student;
    d2rq:property ex:personID;
    d2rq:refersToClassMap map:PersonID_student;
    d2rq:join "XSTUDENT.STUDENT_ID => XSTUDENT1.STUDENT_ID";
    d2rq:alias "XSTUDENT AS XSTUDENT1";
    .
# three property bridges connects Person class maps
# which is for XTeacher table to PersonID class map
# which is for XTeacher table (a shorter version)
map:personID2 a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:Teacher;
    d2rq:property ex:personID;
    d2rq:refersToClassMap map:PersonID_teacher;
    .
```

A.2.2 D2RQ mapping script for far-table-linking example [2.3.2]

```
@prefix d2rq: <http://www.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/0.1#> .
@prefix jdbc: <http://d2rq.org/terms/jdbc/> .
@prefix ex: <http://lumii.lv/ex#> .

map:database a d2rq:Database;
 d2rq:jdbcDriver "oracle.jdbc.driver.OracleDriver";
 d2rq:jdbcDSN "jdbc:oracle:thin:@guntars-PC:1521:gun";
 d2rq:username "far_links";
 d2rq:password "f";
 .
# Something class
map:ClassForTable a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "table@@TABLE1.TABLE1_ID@@";
 d2rq:class ex: Something;
 .
# localName property
 map:localName a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:ClassForTable;
 d2rq:property ex:localName;
 d2rq:column "TABLE1.NAME";
 .
map:farName a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:ClassForTable;
    d2rq:property ex:farName;
    d2rq:join "TABLE1.TABLE2_ID => TABLE2.TABLE2_ID ";
    d2rq:join "TABLE2.TABLE3_ID => TABLE3.TABLE3_ID ";
```

```
    d2rq:join "TABLE3.TABLE4_ID => TABLE4.TABLE4_ID ";
    d2rq:column "TABLE4.NAME";
.
map:farPath a d2rq:PropertyBridge;
    d2rq:belongsToClassMap map:ClassForTable;
    d2rq:property ex:farPath;
    d2rq:join "TABLE1.TABLE2_ID => TABLE2.TABLE2_ID ";
    d2rq:join "TABLE2.TABLE3_ID => TABLE3.TABLE3_ID ";
    d2rq:join "TABLE3.TABLE4_ID => TABLE4.TABLE4_ID ";
    d2rq:sqlExpression "TABLE1.NAME || ' -> ' || TABLE2.NAME
|| ' -> ' || TABLE3.NAME || ' -> ' || TABLE4.NAME";
.
```

A.2.3 D2RQ mapping code for genealogy example [2.3.3]

```
@prefix map: <file:/C:/semantic_web/d2r-server-
0.7/school_mapping.n3#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d2rq: <http://www.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/0.1#> .
@prefix ex: <http://lumii.lv/ex#> .

map:database a d2rq:Database;
 d2rq:jdbcDriver "oracle.jdbc.driver.OracleDriver";
 d2rq:jdbcDSN "jdbc:oracle:thin:@guntars-PC:1521:gun";
 d2rq:username "genealogy";
 d2rq:password "g";
.
# Person class
map:Person a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriPattern "person@@PERSON.PERSON_ID@@";
 d2rq:class ex:Person;
.
# personName property
 map:personName a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:personName;
 d2rq:column "PERSON.NAME";
 #d2rq:datatype xsd:string;
.
# birthYear property
 map:birthYear a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:birthYear;
 d2rq:column "PERSON.BIRTH_YEAR";
 d2rq:datatype xsd:integer;
.
# deathYear property
 map:deathYear a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:deathYear;
 d2rq:column "PERSON.DEATH_YEAR";
 d2rq:datatype xsd:integer;
.
# lifeSpan property
 map:lifeSpan a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
```

```
 d2rq:property ex:lifeSpan;
 d2rq:sqlExpression "PERSON.DEATH_YEAR - PERSON.BIRTH_YEAR";
 d2rq:datatype xsd:integer;
 .
map:parent_father a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:parent;
 d2rq:refersToClassMap map:Person;
 d2rq:alias "PERSON AS PARENT";
 d2rq:join "PERSON.FATHER_ID => PARENT.PERSON_ID ";
    .
map:parent_mother a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:parent;
 d2rq:refersToClassMap map:Person;
 d2rq:alias "PERSON AS PARENT";
 d2rq:join "PERSON.MOTHER_ID => PARENT.PERSON_ID ";
    .
# PersonType class
map:Gender a d2rq:ClassMap;
 d2rq:dataStorage map:database;
 d2rq:uriColumn "PERSON.GENDER";
 d2rq:containsDuplicates "true";
 d2rq:class ex:Gender;
 d2rq:translateWith map:GenderTable
 .
map:GenderTable a d2rq:TranslationTable;
        d2rq:translation [ d2rq:databaseValue "f"; d2rq:rdfValue
"ex:female"; ];
        d2rq:translation [ d2rq:databaseValue "m"; d2rq:rdfValue
"ex:male"; ]
.
map:gender a d2rq:PropertyBridge;
 d2rq:belongsToClassMap map:Person;
 d2rq:property ex:gender;
 d2rq:refersToClassMap map:Gender;
.
```

A.3 Virtuoso RDF Views mapping code for mini-university example [2.3.1]

Ontology mappings:

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix virtrdf: <http://www.openlinksw.com/schemas/virtrdf#> .
@prefix DB: <http://lumiiex/school/> .

DB:Course a rdfs:Class .
DB:courseName a owl:DatatypeProperty .
DB:courseName rdfs:range xsd:string .
DB:courseName rdfs:domain DB:Course .
DB:isTaughtBy a owl:ObjectProperty .
DB:isTaughtBy rdfs:domain DB:Course .
DB:isTaughtBy rdfs:range DB:Teacher .

DB:MandatoryCourse a rdfs:Class .
```

```
DB:MandatoryCourse rdfs:subClassOf DB:Course .

DB:OptionalCourse a rdfs:Class .
DB:OptionalCourse rdfs:subClassOf DB:Course .


DB:Person a rdfs:Class .
DB:personName a owl:DatatypeProperty .
DB:personName rdfs:range xsd:string .
DB:personName rdfs:domain DB:Person .

DB:Teacher a rdfs:Class .
DB:Teacher rdfs:subClassOf DB:Person .
DB:teaches a owl:ObjectProperty .
DB:teaches rdfs:domain DB:Teacher .
DB:teaches rdfs:range DB:Course .

DB:Assistant a rdfs:Class .
DB:Assistant rdfs:subClassOf DB:Teacher .

DB:Professor a rdfs:Class .
DB:Professor rdfs:subClassOf DB:Teacher .

DB:AssocProfessor a rdfs:Class .
DB:AssocProfessor rdfs:subClassOf DB:Teacher .

DB:Student a rdfs:Class .
DB:Student rdfs:subClassOf DB:Person .
DB:takes a owl:ObjectProperty .
DB:takes rdfs:domain DB:Student .
DB:takes rdfs:range DB:Course .

DB:AcademicProgram a rdfs:Class .
DB:programName a owl:DatatypeProperty .
DB:personName rdfs:range xsd:string .
DB:personName rdfs:domain DB:AcademicProgram .
DB:enrolled a owl:ObjectProperty .
DB:enrolled rdfs:domain DB:Student .
DB:enrolled rdfs:range DB:AcademicProgram .
DB:includes a owl:ObjectProperty .
DB:includes rdfs:domain DB:AcademicProgram .
DB:includes rdfs:range DB:Course .
DB:belongsTo a owl:ObjectProperty .
DB:belongsTo rdfs:domain DB:Course .
DB:belongsTo rdfs:range DB:AcademicProgram .

DB:PersonID a rdfs:Class .
DB:IDValue a owl:DatatypeProperty .
DB:IDValue rdfs:range xsd:string .
DB:IDValue rdfs:domain DB:PersonID .
DB:personID a owl:ObjectProperty .
DB:personID rdfs:domain DB:Person .
DB:personID rdfs:range DB:PersonID .
```

Data mappings:

```
grant select on DB.DBA.COURSE to SPARQL_SELECT;
grant select on DB.DBA.TEACHER to SPARQL_SELECT;
grant select on DB.DBA.STUDENT to SPARQL_SELECT;
grant select on DB.DBA.REGISTRATION to SPARQL_SELECT;
```

```
grant select on DB.DBA.PROGRAM to SPARQL_SELECT;

SPARQL
drop quad storage virtrdf:school;


SPARQL
prefix DB: <http://lumiiex/school/>
create iri class DB:teacher_iri "http://lumiiex/school/teacher%d"
(in _TEACHER_ID numeric not null) . ;

SPARQL
prefix DB: <http://lumiiex/school/>
create iri class DB:student_iri "http://lumiiex/school/student%d"
(in _STUDENT_ID numeric not null) . ;

SPARQL
prefix DB: <http://lumiiex/school/>
create iri class DB:program_iri "http://lumiiex/school/program%d"
(in _PROGRAM_ID numeric not null) . ;

SPARQL
prefix DB: <http://lumiiex/school/>
create iri class DB:personID_iri "http://lumiiex/school/personID%s"
(in _IDCODE varchar not null) . ;

SPARQL
prefix DB: <http://lumiiex/school/>
create iri class DB:course_iri "http://lumiiex/school/course%d"
(in _COURSE_ID numeric not null) . ;


SPARQL
# Class Course
prefix DB: <http://lumiiex/school/>
create quad storage virtrdf:school
 from DB.DBA.COURSE as course_s
 from DB.DBA.TEACHER as teacher_s
   where (^{course_s.}^.TEACHER_ID = ^{teacher_s.}^."TEACHER_ID")
{
 create DB:qm-course as graph <http://lumiiex/school/#>
 {
  DB:course_iri(course_s.COURSE_ID) a DB:Course ;
       DB:courseName course_s.NAME ;
       DB:isTaughtBy DB:teacher_iri (teacher_s.TEACHER_ID) .
 }
};

SPARQL
# Class MandatoryCourse
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.COURSE as course_s_mand
   where (^{course_s_mand.}^.REQUIRED = 1)
{
 create DB:qm-mandatory_course as graph <http://lumiiex/school/#>
 {
  DB:course_iri (course_s_mand.COURSE_ID)  a DB:MandatoryCourse .
 }
};
```

```
SPARQL
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.COURSE as course_s0
   where (^{course_s0.}^.REQUIRED = 0)
{
 create DB:qm-optional_course as graph <http://lumiiex/school/#>
 {
  # Maps from columns of "DB.DBA.COURSE"
  DB:course_iri (course_s0."COURSE_ID") a DB:OptionalCourse .
 }
};

SPARQL
# Class Teacher
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.TEACHER as teacher_s
 from DB.DBA.COURSE as course_s
  where (^{course_s.}^.TEACHER_ID = ^{teacher_s.}^.TEACHER_ID)
{
 create DB:qm-teacher as graph  <http://lumiiex/school/#>
 {
  # Maps from columns of "DB.DBA.TEACHER"
  DB:teacher_iri(teacher_s."TEACHER_ID")  a DB:Teacher ;
     DB:personName teacher_s."NAME" as DB:dba-teacher-name ;
     DB:teaches DB:course_iri(course_s."COURSE_ID")   ;
     DB:personID DB:personID_iri(teacher_s.IDCODE)   .

 }
};

SPARQL
# Assistant (subclass of Teacher)
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.TEACHER as teacher_s1
  where (^{teacher_s1.}^.LEVEL_CODE='Assistant')
{
 create DB:qm-teacher_assistant as graph  <http://lumiiex/school/#>
 {
  DB:teacher_iri(teacher_s1.TEACHER_ID)  a DB:Assistant .
 }
};

SPARQL
# Professor (subclass of Teacher)
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.TEACHER as teacher_s2
  where (^{teacher_s2.}^.LEVEL_CODE='Profssor')
{
 create DB:qm-teacher_professor as graph  <http://lumiiex/school/#>
 {
  DB:teacher_iri(teacher_s2.TEACHER_ID)  a DB:Professor .
 }
};
```

```
SPARQL
# AssocProfessor (subclass of Teacher)
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.TEACHER as teacher_s3
  where (^{teacher_s3.}^.LEVEL_CODE='AssocProfessor')
{
 create DB:qm-teacher_assos_professor as graph
<http://lumiiex/school/#>
 {
  DB:teacher_iri(teacher_s3.TEACHER_ID)  a DB:AssocProfessor .
 }
};


SPARQL
# Class Student, object property takes, etc
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.STUDENT as student_s
 from DB.DBA.REGISTRATION as registration_s
  where ( ^{registration_s.}^.STUDENT_ID = ^{student_s.}^.STUDENT_ID
)
 from DB.DBA.COURSE as course_s_taken
  where ( ^{course_s_taken.}^.COURSE_ID =
^{registration_s.}^.COURSE_ID  )
{
 create DB:qm-student as graph  <http://lumiiex/school/#>
 {
  DB:student_iri(student_s.STUDENT_ID)  a DB:Student ;
     DB:personName student_s.NAME as DB:dba-student-name ;
     DB:takes DB:course_iri(registration_s.COURSE_ID)  ;
     DB:personID DB:personID_iri(student_s.IDCODE)   .
 }
};


SPARQL
# Class AcademicProgram
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.PROGRAM as program_s
 from DB.DBA.COURSE as course_s_included
  where (^{course_s_included.}^.PROGRAM_ID =
^{program_s.}^.PROGRAM_ID)
{
 create DB:qm-program as graph  <http://lumiiex/school/#>
 {
  DB:program_iri(program_s.PROGRAM_ID)  a DB:AcademicProgram ;
     DB:programName program_s.NAME as DB:dba-program-name ;
     DB:includes DB:course_iri(course_s_included.COURSE_ID)   .
 }
};


SPARQL
# Class PersonID 1. map
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.TEACHER as teacher_s_as_person
```

```
{
 create DB:qm-teacher-person as graph  <http://lumiiex/school/#>
 {
  DB:personID_iri(teacher_s_as_person.IDCODE) a DB:PersonID ;
     DB:IDValue teacher_s_as_person.IDCODE  .
 }
};

SPARQL
# Class PersonID 2. map
prefix DB: <http://lumiiex/school/>
alter quad storage virtrdf:school
 from DB.DBA.STUDENT as student_s_as_person
{
 create DB:qm-student-person as graph  <http://lumiiex/school/#>
 {
  DB:personID_iri(student_s_as_person.IDCODE) a DB:PersonID ;
     DB:IDValue student_s_as_person.IDCODE  .
 }
};
```

A.4 D2O mapping code for mini-university example [2.3.1]

```
<?xml version="1.0" encoding="UTF-8"?>
<r2o>
 <dbschema-desc name="db">
  <has-table name="PROGRAM">
   <keycol-desc name="PROGRAM_ID"/>
   <nonkeycol-desc name="NAME"/>
  </has-table>
  <has-table name="STUDENT">
   <keycol-desc name="STUDENT_ID"/>
   <forkeycol-desc name="PROGRAM_ID">
    <refers-to>PROGRAM.PROGRAM_ID</refers-to>
   </forkeycol-desc>
   <nonkeycol-desc name="NAME"/>
   <nonkeycol-desc name="IDCODE"/>
  </has-table>
  <has-table name="COURSE">
   <keycol-desc name="COURSE_ID"/>
   <forkeycol-desc name="TEACHER_ID">
    <refers-to>TEACHER.TEACHER_ID</refers-to>
   </forkeycol-desc>
   <forkeycol-desc name="PROGRAM_ID">
    <refers-to>PROGRAM.PROGRAM_ID</refers-to>
   </forkeycol-desc>
   <nonkeycol-desc name="NAME"/>
   <nonkeycol-desc name="REQUIRED"/>
  </has-table>
  <has-table name="TEACHER">
   <keycol-desc name="TEACHER_ID"/>
   <forkeycol-desc name="LEVEL_CODE">
    <refers-to>TEACHER_LEVEL.LEVEL_CODE</refers-to>
   </forkeycol-desc>
   <nonkeycol-desc name="NAME"/>
   <nonkeycol-desc name="IDCODE"/>
  </has-table>
```

```
  <has-table name="TEACHER_LEVEL">
   <keycol-desc name="LEVEL_CODE"/>
  </has-table>
  <has-table name="REGISTRATION">
   <keycol-desc name="REGISTRATION_ID"/>
   <forkeycol-desc name="COURSE_ID">
    <refers-to>COURSE.COURSE_ID</refers-to>
   </forkeycol-desc>
   <forkeycol-desc name="STUDENT_ID">
    <refers-to>STUDENT.STUDENT_ID</refers-to>
   </forkeycol-desc>
  </has-table>
 </dbschema-desc>
 <conceptmap-def name="http://lumii.lv/ex#Student">
  <uri-as >
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#Student</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.STUDENT.STUDENT_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <described-by>
   <attributemap-def name="http://lumii.lv/ex#personName">
    <selector>
     <aftertransform>
      <operation oper-id="concat">
       <arg-restriction on-param="string1">
        <has-column>db.STUDENT.NAME</has-column>
       </arg-restriction>
       <arg-restriction on-param="string2">
        <has-transform>
         <operation oper-id="concat">
          <arg-restriction on-param="string1">
           <has-value>" "</has-value>
          </arg-restriction>
          <arg-restriction on-param="string2">
           <has-column>db.STUDENT.SURNAME</has-column>
          </arg-restriction>
         </operation>
        </has-transform>
       </arg-restriction>
      </operation>
     </aftertransform>
    </selector>
   </attributemap-def>
   <dbrelationmap-def name="http://lumii.lv/ex#enrolled"
toConcept="http://lumii.lv/ex#AcademicProgram">
    <joins-via>
     <condition oper-id="equals">
      <arg-restriction on-param="value1">
       <has-column>db.STUDENT.PROGRAM_ID</has-column>
      </arg-restriction>
      <arg-restriction on-param="value2">
       <has-column>db.PROGRAM.PROGRAM_ID</has-column>
      </arg-restriction>
     </condition>
    </joins-via>
```

```
   </dbrelationmap-def>
   <dbrelationmap-def name="http://lumii.lv/ex#takes"
toConcept="http://lumii.lv/ex#Course">
    <joins-via>
     <AND>
      <condition oper-id="equals">
       <arg-restriction on-param="value1">
        <has-column>db.STUDENT.STUDENT_ID</has-column>
       </arg-restriction>
       <arg-restriction on-param="value2">
        <has-column>db.REGISTRATION.STUDENT_ID</has-column>
       </arg-restriction>
      </condition>
      <condition oper-id="equals">
       <arg-restriction on-param="value1">
        <has-column>db.REGISTRATION.COURSE_ID</has-column>
       </arg-restriction>
       <arg-restriction on-param="value2">
        <has-column>db.COURSE.COURSE_ID</has-column>
       </arg-restriction>
      </condition>
     </AND>
    </joins-via>
   </dbrelationmap-def>
  </described-by>
 </conceptmap-def>
 <conceptmap-def name="http://lumii.lv/ex#Teacher">
  <uri-as>
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#Teacher</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.TEACHER.TEACHER_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <described-by>
   <attributemap-def name="http://lumii.lv/ex#personName">
    <selector>
     <aftertransform>
      <operation oper-id="constant">
       <arg-restriction on-param="const-val">
        <has-column>db.TEACHER.NAME</has-column>
       </arg-restriction>
      </operation>
     </aftertransform>
    </selector>
   </attributemap-def>
   <dbrelationmap-def name="http://lumii.lv/ex#teaches"
toConcept="http://lumii.lv/ex#Course">
    <joins-via>
     <condition oper-id="equals">
      <arg-restriction on-param="value1">
       <has-column>db.TEACHER.TEACHER_ID</has-column>
      </arg-restriction>
      <arg-restriction on-param="value2">
       <has-column>db.COURSE.TEACHER_ID</has-column>
      </arg-restriction>
     </condition>
```

```
    </joins-via>
   </dbrelationmap-def>
  </described-by>
 </conceptmap-def>
 <conceptmap-def name="http://lumii.lv/ex#Asistant">
  <uri-as>
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#Teacher</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.TEACHER.TEACHER_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <applies-if>
   <condition oper-id="equals">
    <arg-restriction on-param="value1">
     <has-column>db.TEACHER.LEVEL_CODE</has-column>
    </arg-restriction>
    <arg-restriction on-param="value2">
     <has-value>"Assistant"</has-value>
    </arg-restriction>
   </condition>
  </applies-if>
 </conceptmap-def>
 <conceptmap-def name="http://lumii.lv/ex#Professor">
  <uri-as>
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#Teacher</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.TEACHER.TEACHER_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <applies-if>
   <condition oper-id="equals">
    <arg-restriction on-param="value1">
     <has-column>db.TEACHER.LEVEL_CODE</has-column>
    </arg-restriction>
    <arg-restriction on-param="value2">
     <has-value>"Professor"</has-value>
    </arg-restriction>
   </condition>
  </applies-if>
 </conceptmap-def>
 <conceptmap-def name="http://lumii.lv/ex#AssocProfessor">
  <uri-as>
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#Teacher</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.TEACHER.TEACHER_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <applies-if>
```

```
  <condition oper-id="equals">
   <arg-restriction on-param="value1">
    <has-column>db.TEACHER.LEVEL_CODE</has-column>
   </arg-restriction>
   <arg-restriction on-param="value2">
    <has-value>"Associate Professor"</has-value>
   </arg-restriction>
  </condition>
 </applies-if>
</conceptmap-def>
<conceptmap-def name="http://lumii.lv/ex#Course">
 <uri-as>
  <operation oper-id="concat">
   <arg-restriction on-param="string1">
    <has-value>http://lumii.lv/ex#Course</has-value>
   </arg-restriction>
   <arg-restriction on-param="string2">
    <has-column>db.COURSE.COURSE_ID</has-column>
   </arg-restriction>
  </operation>
 </uri-as>
 <described-by>
  <attributemap-def name="http://lumii.lv/ex#courseName">
   <selector>
    <aftertransform>
     <operation oper-id="constant">
      <arg-restriction on-param="const-val">
       <has-column>db.COURSE.NAME</has-column>
      </arg-restriction>
     </operation>
    </aftertransform>
   </selector>
  </attributemap-def>
 </described-by>
</conceptmap-def>
<conceptmap-def name="http://lumii.lv/ex#MandatoryCourse">
 <uri-as>
  <operation oper-id="concat">
   <arg-restriction on-param="string1">
    <has-value>http://lumii.lv/ex#MandatoryCourse</has-value>
   </arg-restriction>
   <arg-restriction on-param="string2">
    <has-column>db.COURSE.COURSE_ID</has-column>
   </arg-restriction>
  </operation>
 </uri-as>
 <applies-if>
  <condition oper-id="equals">
   <arg-restriction on-param="value1">
    <has-column>db.COURSE.REQUIRED</has-column>
   </arg-restriction>
   <arg-restriction on-param="value2">
    <has-value>1</has-value>
   </arg-restriction>
  </condition>
 </applies-if>
</conceptmap-def>
<conceptmap-def name="http://lumii.lv/ex#OptionalCourse">
 <uri-as type="DEFAULT">
  <operation>
```

```
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#OptionalCourse</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.COURSE.COURSE_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <applies-if>
   <condition oper-id="equals">
    <arg-restriction on-param="value1">
     <has-column>db.COURSE.REQUIRED</has-column>
    </arg-restriction>
    <arg-restriction on-param="value2">
     <has-value>0</has-value>
    </arg-restriction>
   </condition>
  </applies-if>
 </conceptmap-def>
 <conceptmap-def name="http://lumii.lv/ex#AcademicProgram">
  <uri-as>
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#AcademicProgram</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.PROGRAM.PROGRAM_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <described-by>
   <attributemap-def name="http://lumii.lv/ex#programName">
    <selector>
     <aftertransform>
      <operation oper-id="constant">
       <arg-restriction on-param="const-val">
        <has-column>db.PROGRAM.NAME</has-column>
       </arg-restriction>
      </operation>
     </aftertransform>
    </selector>
   </attributemap-def>
  </described-by>
 </conceptmap-def>
 <conceptmap-def name="http://lumii.lv/ex#PersonID">
  <uri-as>
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#PersonID</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.STUDENT.STUDENT_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <described-by>
   <attributemap-def name="http://lumii.lv/ex#IDValue">
    <selector>
     <aftertransform>
      <operation oper-id="constant">
```

```
        <arg-restriction on-param="const-val">
         <has-column>db.STUDENT.IDCODE</has-column>
        </arg-restriction>
       </operation>
      </aftertransform>
     </selector>
    </attributemap-def>
   </described-by>
  </conceptmap-def>
  <conceptmap-def name="http://lumii.lv/ex#PersonID">
   <uri-as>
    <operation oper-id="concat">
     <arg-restriction on-param="string1">
      <has-value>http://lumii.lv/ex#PersonID</has-value>
     </arg-restriction>
     <arg-restriction on-param="string2">
      <has-column>db.TEACHER.TEACHER_ID</has-column>
     </arg-restriction>
    </operation>
   </uri-as>
   <described-by>
    <attributemap-def name="http://lumii.lv/ex#IDValue">
     <selector>
      <aftertransform>
       <operation oper-id="constant">
        <arg-restriction on-param="const-val">
         <has-column>db.TEACHER.IDCODE</has-column>
        </arg-restriction>
       </operation>
      </aftertransform>
     </selector>
    </attributemap-def>
   </described-by>
  </conceptmap-def>
  <conceptmap-def name="http://lumii.lv/ex#Person">
   <identified-by>db.TEACHER.TEACHER_ID</identified-by>
   <uri-as>
    <operation oper-id="concat">
     <arg-restriction on-param="string1">
      <has-value>http://lumii.lv/ex#PersonID</has-value>
     </arg-restriction>
     <arg-restriction on-param="string2">
      <has-column>db.TEACHER.TEACHER_ID</has-column>
     </arg-restriction>
    </operation>
   </uri-as>
   <described-by>
    <attributemap-def name="http://lumii.lv/ex#IDValue">
     <selector>
      <aftertransform>
       <operation oper-id="constant">
        <arg-restriction on-param="const-val">
         <has-column>db.TEACHER.IDCODE</has-column>
        </arg-restriction>
       </operation>
      </aftertransform>
     </selector>
    </attributemap-def>
    <dbrelationmap-def name="http://lumii.lv/ex#personID"
toConcept="http://lumii.lv/ex#PersonID">
```

```
    <joins-via>
     <!-- Problem to make join as both Classes for domain and range
uses the same table.
      R2O language does not has facilities to assign aliases  to
tables
     -->
     <condition oper-id="equals">
      <arg-restriction on-param="value1">
       <has-column>db.TEACHER.TEACHER_ID</has-column>
      </arg-restriction>
      <arg-restriction on-param="value2">
       <has-column>db.TEACHER.TEACHER_ID</has-column>
      </arg-restriction>
     </condition>
    </joins-via>
   </dbrelationmap-def>
  </described-by>
 </conceptmap-def>
 <conceptmap-def name="http://lumii.lv/ex#Person">
  <identified-by>db.STUDENT.STUDENT_ID</identified-by>
  <uri-as>
   <operation oper-id="concat">
    <arg-restriction on-param="string1">
     <has-value>http://lumii.lv/ex#PersonID</has-value>
    </arg-restriction>
    <arg-restriction on-param="string2">
     <has-column>db.STUDENT.STUDENT_ID</has-column>
    </arg-restriction>
   </operation>
  </uri-as>
  <described-by>
   <attributemap-def name="http://lumii.lv/ex#IDValue">
    <selector>
     <aftertransform>
      <operation oper-id="constant">
       <arg-restriction on-param="const-val">
        <has-column>db.STUDENT.IDCODE</has-column>
       </arg-restriction>
      </operation>
     </aftertransform>
    </selector>
   </attributemap-def>
   <dbrelationmap-def name="http://lumii.lv/ex#personID"
toConcept="http://lumii.lv/ex#PersonID">
    <joins-via>
     <!-- Problem to make join as both Classes for domain and range
uses the same table.
      R2O language does not has facilities to assign aliases  to
tables
     -->
     <condition oper-id="equals">
      <arg-restriction on-param="value1">
       <has-column>db.STUDENT.STUDENT_ID</has-column>
      </arg-restriction>
      <arg-restriction on-param="value2">
       <has-column>db.STUDENT.STUDENT_ID</has-column>
      </arg-restriction>
     </condition>
    </joins-via>
   </dbrelationmap-def>
```

```
  </described-by>
 </conceptmap-def>
</r2o>
```

A.5 R2RML mapping code for mini-university example

TODO: some comments.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://lumii.lv/ex#>.


<#TriplesMap_Program>
    a rr:TriplesMapClass;
    rr:tableOwner "SCHOOL";
    rr:tableName  "PROGRAM";

    rr:subjectMap [ rr:template "ex:program{program_id}";
                    rr:class ex:Program;
                  ];

    rr:predicateObjectMap
    [
      rr:predicateMap [ rr:predicate ex:programName ];
      rr:objectMap    [ rr:column "name"  ]
    ];

    rr:refPredicateObjectMap
    [
        rr:refPredicateMap [  rr:predicate ex:includes ];
        rr:refObjectMap
        [
            rr:parentTriplesMap <#TriplesMap_Course>;
            rr:joinCondition
                "{childAlias.}program_id = {parentAlias.}program_id"
        ]
    ]
    .

<#TriplesMap_Course>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  course_id
            , teacher_id
            , program_id
            , name
            , case when required=1 then 'MandatoryCourse'
                   else 'OptionalCourse'
              end as subclass_name
      from    COURSE
    """;

    rr:subjectMap [ rr:template "ex:course{course_id}";
                    rr:class ex:Course;
                  ];
```

```
    rr:predicateObjectMap
    [
      rr:predicateMap [ rr:predicate rdf:type ];
      rr:objectMap    [ rr:template "ex:{subclass_name}"   ]
    ];

    rr:predicateObjectMap
    [
      rr:predicateMap [ rr:predicate ex:courseName ];
      rr:objectMap    [ rr:column "name" ]
    ];
    .

    <#PredicateObjectMap_personName>
    a rr:PredicateObjectMapClass
    [
      rr:predicateMap [ rr:predicate ex:personName ];
      rr:objectMap    [ rr:column "name"   ]
    ];
    .

<#TriplesMap_Teacher>
    a rr:TriplesMapClass;
    rr:tableOwner "SCHOOL";
    rr:tableName  "TEACHER";

    rr:subjectMap [ rr:template "ex:teacher{teacher_id}";
                    rr:class ex:Teacher;
                  ];

    rr:predicateObjectMap <#PredicateObjectMap_personName> ;

    rr:refPredicateObjectMap
    [
        rr:refPredicateMap [  rr:predicate ex:teaches ];
        rr:refObjectMap
        [
            rr:parentTriplesMap <#TriplesMap_Course>;
            rr:joinCondition
                "{childAlias.}teacher_id = {parentAlias.}teacher_id"
        ]
    ]

    rr:refPredicateObjectMap
    [
        rr:refPredicateMap [  rr:predicate ex:personID ];
        rr:refObjectMap
        [
            rr:parentTriplesMap <#TriplesMap_PersonID_teacher>;
            rr:joinCondition
                "{childAlias.}teacher_id = {parentAlias.}teacher_id"
        ]
    ]
    .


<#TriplesMap_Assistant>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  teacher_id
```

```
          , name
      from    TEACHER
      where   level_code='Assistant'
    """;

    rr:subjectMap [ rr:template "ex:teacher{teacher_id}";
                    rr:class ex:Assistant;
                  ];

    rr:predicateObjectMap <#PredicateObjectMap_personName> ;
    .

<#TriplesMap_Professor>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  teacher_id
            , name
      from    TEACHER
      where   level_code='Professor'
    """;

    rr:subjectMap [ rr:template "ex:teacher{teacher_id}";
                    rr:class ex:Professor;
                  ];

    rr:predicateObjectMap <#PredicateObjectMap_personName> ;
    .


<#TriplesMap_AssocProfessor>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  teacher_id
            , name
      from    TEACHER
      where   level_code='Associate Profssor'
    """;

    rr:subjectMap [ rr:template "ex:teacher{teacher_id}";
                    rr:class ex:AssocProfessor;
                  ];

    rr:predicateObjectMap <#PredicateObjectMap_personName>;
    .



<#TriplesMap_Student>
    a rr:TriplesMapClass;
    rr:SQLQuery """
      Select  s.student_id
            , s.program_id
            , s.name
            , r.course_id
      from    STUDENT s, REGISTRATION r
      where   s.student_id=r.student_id
    """;

    rr:subjectMap [ rr:template "ex:student{student_id}";
                    rr:class ex:Student;
```

```
                ];

    rr:predicateObjectMap <#PredicateObjectMap_personName>;

    rr:refPredicateObjectMap
    [
        rr:refPredicateMap [  rr:predicate ex:enrolled ];
        rr:refObjectMap
        [
            rr:parentTriplesMap <#TriplesMap_Program>;
            rr:joinCondition
                "{childAlias.}program_id = {parentAlias.}program_id"
        ]
    ] ;

    rr:refPredicateObjectMap
    [
        rr:refPredicateMap [  rr:predicate ex:takes ];
        rr:refObjectMap
        [
            rr:parentTriplesMap <#TriplesMap_Course>;
            rr:joinCondition
                "{childAlias.}course_id = {parentAlias.}course_id"
        ]
    ]

    rr:refPredicateObjectMap
    [
        rr:refPredicateMap [  rr:predicate ex:personID ];
        rr:refObjectMap
        [
            rr:parentTriplesMap <#TriplesMap_PersonID_student>;
            rr:joinCondition
                "{childAlias.}student_id = {parentAlias.}student_id"
        ]
    ]
    .

<#TriplesMap_PersonID_teacher>
    a rr:TriplesMapClass;
    rr:tableOwner "SCHOOL";
    rr:tableName  "TEACHER";

    rr:subjectMap [ rr:template "ex:personID{idcode}";
                    rr:class ex:PersonID;
                  ];

    rr:predicateObjectMap
    [
      rr:predicateMap [ rr:predicate ex:IDValue ];
      rr:objectMap    [ rr:column "idcode" ]
    ];
    .

<#TriplesMap_PersonID_student>
    a rr:TriplesMapClass;
    rr:tableOwner "SCHOOL";
    rr:tableName  "STUDENT";

    rr:subjectMap [ rr:template "ex:personID{idcode}";
```

```
                rr:class ex:PersonID;
              ];

    rr:predicateObjectMap
    [
      rr:predicateMap [ rr:predicate ex:IDValue ];
      rr:objectMap    [ rr:column "idcode" ]
    ];
    .
```

A.6 RDB2OWL SQL codes for tripple generations

We provide listings of SQL scripts which when executed in *mapping DB* generate SQL scripts which in turn when executed in *source DB* generate RDF triples for instances of OWL classes, OWL datatype properties and OWL object properties.

They are not easily readable as two SQL levels are mixed. They show that mere SQL statement can do the task. They generate SQL statements using string concatenation operator + as it is in MsSQL Server (Standard SQL uses || for this). They also use MsSQL functions ISNULL, REPLACE and others. It is easy to rewrite these SQLs for another DB if needed.

A.6.1 SQL code for class instance generation

SQL script ***OWL_instance_gen.sql*** that generates SQL statement for RDF triple generation for OWL class instances

```
SELECT
  'SELECT '
  + '''<' + o.xml_base
  +  cm.instance_uri_prefix + ''''
  + ' + '
  + 'CAST('
  + (CASE WHEN t.is_column_expr=1 THEN ' ' ELSE 't.' END)
  + cm.id_column_expr + ' AS varchar) + ''>'' as subject'
  + ',''<' + o.type_uri + '>'' as predicate'
  + ',''' + '<' + o.xml_base + c.rdf_id + '>'' as object'
  + ' FROM '
  + t.table_name + ' t '
  + (CASE WHEN cm.filter_expr IS NULL THEN ' ' ELSE ' WHERE ' END) +
ISNULL(cm.filter_expr,'') as sql4rdf
FROM
  ontology o,  owl_class c, class_map cm, db_table t
WHERE
  o.ontology_id = c.ontology_id AND
  c.owl_class_id = cm.owl_class_id AND
  cm.db_table_id = t.db_table_id AND
  cm.id_column_expr IS NOT NULL AND
  LEN(cm.id_column_expr)>0 AND
  o.ontology_id=1 AND  cm.generate_instances=1
```

A.6.2 SQL code for data property value generation

SQL script ***generate_sql4datatype_props.sql*** that generates SQL statements for RDF triple generation for OWL datatype property instances

```
SELECT
  'SELECT '
  + '''<' + o.xml_base
  +  cm.instance_uri_prefix + ''''
  + ' + CAST(' +
  + (CASE WHEN t.is_column_expr=1 THEN ' ' ELSE 't.' END)
  +  cm.id_column_expr + ' AS VARCHAR) + ''>'' as subject'
  + ','''' + '<'
  + o.xml_base + dp.rdf_id  + '>'' as predicate'
  + ', '''"''+ '
  + CASE WHEN sqld.type_name IN ('varchar','char','nvarchar','nchar')
THEN '' ELSE ' CAST(' END
  + CASE WHEN tl.mid_table_id IS NULL THEN
    ''
  ELSE
    CASE WHEN dpm.is_column_expr=1 THEN '' ELSE 't_link.' END
  END
  + REPLACE( REPLACE( dpm.column_expr, 't.','t_link.' ), 's.', 't.' )
  + CASE WHEN sqld.type_name IN ('varchar','char','nvarchar','nchar')
THEN '' ELSE ' AS varchar)'  END
  +'+ '''"^^xsd:'
  + ISNULL(xsdd.type_name,'string')
  + ''' as object'
  + ' FROM '
  + t.table_name + ' t '
  + CASE WHEN
 tl.mid_table_id IS NOT NULL AND
 tl.source_column_expr IS NOT NULL AND
 dpm.source_column_expr IS NOT NULL
    THEN
       'INNER JOIN ' + t_tl.table_name + ' t_link ON t.'
      + dpm.source_column_expr + ' = t_link.' + tl.source_column_expr
    ELSE
      ''
    END
  + CASE WHEN
 tl.mid_table_id IS NOT NULL AND
 tl.source_column_expr IS NULL AND     dpm.source_column_expr IS NULL
AND
 tl.filter_expr IS NOT NULL
    THEN
      'INNER JOIN ' + t_tl.table_name + ' t_link ON '
      + REPLACE( REPLACE(tl.filter_expr, 't.','t_link.'), 's.', 't.')
    ELSE
      ''
    END
  + ' WHERE ' + REPLACE( REPLACE( dpm.column_expr, 't.','t_link.' ),
's.', 't.' ) + ' IS NOT NULL '
  + CASE WHEN cm.filter_expr IS NULL THEN ' ' ELSE ' AND ' END
  + ISNULL(cm.filter_expr,'') as sql4rdf,
   ISNULL(xsdd.type_name,'-')
FROM
  ontology o
```

```
  INNER JOIN owl_datatype_property dp ON dp.ontology_id=o.ontology_id
  INNER JOIN datatype_property_map dpm ON
  dp.owl_datatype_property_id=dpm.owl_datatype_property_id
  LEFT OUTER JOIN table_link tl ON dpm.table_link_id=tl.table_link_id
  LEFT OUTER JOIN db_table t_tl ON t_tl.db_table_id=tl.mid_table_id
  INNER JOIN class_map cm ON dpm.class_map_id = cm.class_map_id
  INNER JOIN db_table t ON cm.db_table_id = t.db_table_id
  LEFT OUTER JOIN db_column col ON col.db_table_id=t.db_table_id
    AND UPPER(col.column_name)=UPPER(dpm.column_expr)
  LEFT OUTER JOIN sql_datatype sqld
    ON sqld.sql_datatype_id=col.sql_datatype_id
  LEFT OUTER JOIN xsd_datatype xsdd ON xsdd.xsd_datatype_id=
    CASE WHEN dpm.xsd_datatype_id IS NULL THEN
      sqld.xsd_datatype_id
    ELSE
      dpm.xsd_datatype_id
    END
WHERE
  o.ontology_id=1 AND
  cm.id_column_expr IS NOT NULL AND
  LEN(cm.id_column_expr)>0
```

A.6.3 SQL code for object property value generation without linked tables

SQL script ***generate_sql4object_props.sql*** that generates SQL statements for RDF triple generation for OWL object property instances without intermediate table link usage

```
SELECT
  'SELECT '
  + '''<' + o.xml_base
  +  cm_source.instance_uri_prefix + ''''
  + ' + CAST('
  + (CASE WHEN t_source.is_column_expr=1
     THEN
       ' '
     ELSE
       't_domain.'
     END)
  + REPLACE(cm_source.id_column_expr,'t.','t_domain.')
  + ' AS varchar) + ''>'' as subject'
  + ','''' + '<'
  + o.xml_base + op.rdf_id  + '>'' as predicate'

  + ','
  + '''<' + o.xml_base
  +  cm_target.instance_uri_prefix + ''''
  + ' + CAST('
  + (CASE WHEN t_target.is_column_expr=1
     THEN
       ' '
     ELSE
       't_range.'
     END)
  + REPLACE(cm_target.id_column_expr, 't.', 't_range.')
  + ' AS varchar) + ''>'' as object'

  + ' FROM '
```

```
  + t_source.table_name + ' t_domain '
  + ' INNER JOIN '
  + t_target.table_name + '  t_range ON '
  + (CASE WHEN opm.source_column_expr IS NOT NULL
       AND opm.target_column_expr IS NOT NULL
     THEN
       ' t_domain.' + opm.source_column_expr + ' = t_range.'
       + opm.target_column_expr
     ELSE
      ''
     END)
  + (CASE WHEN opm.source_column_expr IS NOT NULL
       AND opm.target_column_expr IS NOT NULL
       AND opm.filter_expr IS NOT NULL
     THEN
       ' AND '
     ELSE
       ''
     END)

  + (CASE WHEN opm.filter_expr IS NOT NULL
     THEN
       REPLACE( REPLACE(opm.filter_expr,'s.', ' t_domain.'),
       't.', 't_range.')
     ELSE
       ''
     END)
  + ' WHERE ' + REPLACE(
ISNULL(cm_source.filter_expr,'1=1'),'t.','t_domain.' )
  + ' AND    ' + REPLACE(
ISNULL(cm_target.filter_expr,'1=1'),'t.','t_domain.' )
  AS generated_SQL
FROM
  owl_object_property op,
  ontology o,
  object_property_map opm,
  class_map cm_source,
  class_map cm_target,
  db_table t_source,
  db_table t_target
WHERE
  op.ontology_id=o.ontology_id AND
  op.owl_object_property_id=opm.owl_object_property_id AND
  opm.source_class_map_id =cm_source.class_map_id AND
  opm.target_class_map_id =cm_target.class_map_id AND
  cm_source.db_table_id=t_source.db_table_id AND
  cm_target.db_table_id=t_target.db_table_id AND
  opm.table_link_id IS NULL AND op.ontology_id=1
ORDER BY 1
```

A.6.4 SQL code for object property value generation with 1 linked table

SQL script ***generate_sql4object_props_table_links.sql*** that generates SQL statements for RDF triple generation for OWL object property instances with one intermediate table link usage

```
SELECT
  d.database_id,
```

```
'SELECT '
+ '''<' + o.xml_base
+  cm_source.instance_uri_prefix + ''''
+ ' + CAST('
+  (CASE WHEN t_source.is_column_expr=1
    THEN ' '
    ELSE 't_domain.'
    END)
+   REPLACE(cm_source.id_column_expr,'t.','t_domain.')
+ ' AS varchar) + ''>'' as subject'
+ ','''' + '<'
+ o.xml_base + op.rdf_id  + '>'' as predicate'
+ ','

+ '''<' + o.xml_base
+  cm_target.instance_uri_prefix + ''''
+ ' + CAST('
+ (CASE WHEN t_target.is_column_expr=1
   THEN ' '
   ELSE 't_range.'
   END)
+ REPLACE(cm_target.id_column_expr, 't.', 't_range.')
+ ' AS varchar) + ''>'' as object'


+ ' FROM '
+ t_source.table_name + ' t_domain '
+ ' INNER JOIN '
+ t_mid.table_name + ' mid1 ON '
+ (CASE WHEN opm.source_column_expr IS NOT NULL
     AND tl.source_column_expr IS NOT NULL
   THEN
     ' t_domain.' + opm.source_column_expr
     + ' = mid1.' + tl.source_column_expr
   ELSE
     ''
   END)
+ (CASE WHEN opm.source_column_expr IS NOT NULL
     AND tl.source_column_expr IS NOT NULL
     AND opm.filter_expr IS NOT NULL
   THEN
    ' AND '
   ELSE
    ''
   END)
+ (CASE WHEN opm.filter_expr IS NOT NULL
   THEN
     REPLACE(
       REPLACE(opm.filter_expr,'s.', ' t_domain.'),
       't.', 't_range.')
   ELSE
     ''
   END)


+ ' INNER JOIN '
+ t_target.table_name + '  t_range ON '
+ (CASE WHEN tl.target_column_expr IS NOT NULL
     AND opm.target_column_expr IS NOT NULL
   THEN
```

```
      ' mid1.' + tl.target_column_expr
      + ' = t_range.' + opm.target_column_expr
    ELSE
      ''
    END)
  + (CASE WHEN tl.target_column_expr IS NOT NULL
      AND opm.target_column_expr IS NOT NULL
      AND tl.filter_expr IS NOT NULL
    THEN
     ' AND '
    ELSE
      ''
    END)
  + (CASE WHEN tl.filter_expr IS NOT NULL
    THEN
      REPLACE(
        REPLACE(opm.filter_expr,'s.', ' mid1.'),
       't.', 't_range.')
    ELSE
      ''
    END)


  + ' WHERE '
  + REPLACE( ISNULL(cm_source.filter_expr,'1=1'),'t.','t_domain.' )
  + ' AND    '
  + REPLACE( ISNULL(cm_target.filter_expr,'1=1'),'t.','t_domain.' )
  AS generated_SQL
FROM
  owl_object_property op,
  ontology o,
  object_property_map opm,
  class_map cm_source,
  class_map cm_target,
  db_table t_source,
  db_table t_target,
  db_database d,
  table_link tl,
  db_table t_mid
WHERE
  op.ontology_id=o.ontology_id AND
  op.owl_object_property_id=opm.owl_object_property_id AND
  opm.source_class_map_id =cm_source.class_map_id AND
  opm.target_class_map_id =cm_target.class_map_id AND
  cm_source.db_table_id=t_source.db_table_id AND
  cm_target.db_table_id=t_target.db_table_id AND
  d.database_id=t_source.database_id AND
  tl.table_link_id=opm.table_link_id AND
  t_mid.db_table_id=tl.mid_table_id
```

A.6.5 SQL code for object property value generation with 2 linked table

SQL script ***generate_sql4object_props_table_links2.sql*** that generates SQL statements for RDF triple generation for OWL object property instances with two intermediate table link usage linking them by:

opm.source_column_expr=tl1.source_column_expr
and
tl1.target_column_expr=tl2.source_column_expr
and
tl2.target_column_expr=opm.target_column_expr

```
SELECT
  d.database_id,
  'SELECT '
  + '''<' + o.xml_base
  +  cm_source.instance_uri_prefix + ''''
  + ' + CAST('
  +  (CASE WHEN t_source.is_column_expr=1
      THEN ' '
      ELSE 't_domain.'
      END)
  +   REPLACE(cm_source.id_column_expr,'t.','t_domain.')
  + ' AS varchar) + ''>'' as subject'
  + ','''' + '<'
  + o.xml_base + op.rdf_id  + '>'' as predicate'
  + ','

  + '''<' + o.xml_base
  +  cm_target.instance_uri_prefix + ''''
  + ' + CAST('
  + (CASE WHEN t_target.is_column_expr=1
     THEN ' '
     ELSE 't_range.'
     END)
  + REPLACE(cm_target.id_column_expr, 't.', 't_range.')
  + ' AS varchar) + ''>'' as object'


  + ' FROM '
  + t_source.table_name + ' t_domain '
  + ' INNER JOIN '
  + t_mid1.table_name + ' mid1 ON '
  + (CASE WHEN opm.source_column_expr IS NOT NULL
       AND tl1.source_column_expr IS NOT NULL
     THEN
       ' t_domain.' + opm.source_column_expr
       + ' = mid1.' + tl1.source_column_expr
     ELSE
       ''
     END)
  + (CASE WHEN opm.source_column_expr IS NOT NULL
       AND tl1.source_column_expr IS NOT NULL
```

```
     AND opm.filter_expr IS NOT NULL
   THEN
    ' AND '
   ELSE
    ''
   END)
+ (CASE WHEN opm.filter_expr IS NOT NULL
   THEN
     REPLACE(
       REPLACE(opm.filter_expr,'s.', ' t_domain.'),
       't.', 't_range.')
   ELSE
     ''
   END)

+ ' INNER JOIN '
+ t_mid2.table_name + ' mid2 ON '
+ (CASE WHEN tl1.source_column_expr IS NOT NULL
     AND tl2.source_column_expr IS NOT NULL
   THEN
     ' t_domain.' + tl1.source_column_expr
     + ' = mid1.' + tl2.source_column_expr
   ELSE
     ''
   END)
+ (CASE WHEN tl1.source_column_expr IS NOT NULL
     AND tl2.source_column_expr IS NOT NULL
     AND tl1.filter_expr IS NOT NULL
   THEN
    ' AND '
   ELSE
    ''
   END)
+ (CASE WHEN tl1.filter_expr IS NOT NULL
   THEN
     REPLACE(
       REPLACE(tl1.filter_expr,'s.', ' t_domain.'),
       't.', 't_range.')
   ELSE
     ''
   END)

+ ' INNER JOIN '
+ t_target.table_name + '  t_range ON '
+ (CASE WHEN tl2.target_column_expr IS NOT NULL
     AND opm.target_column_expr IS NOT NULL
   THEN
     ' mid2.' + tl2.target_column_expr
     + ' = t_range.' + opm.target_column_expr
   ELSE
     ''
   END)
+ (CASE WHEN tl2.target_column_expr IS NOT NULL
     AND opm.target_column_expr IS NOT NULL
     AND tl2.filter_expr IS NOT NULL
   THEN
    ' AND '
   ELSE
     ''
   END)
```

```
  + (CASE WHEN tl2.filter_expr IS NOT NULL
     THEN
       REPLACE(
         REPLACE(tl2.filter_expr,'s.', ' mid1.'),
        't.', 't_range.')
     ELSE
       ''
     END)


  + ' WHERE '
  + REPLACE( ISNULL(cm_source.filter_expr,'1=1'),'t.','t_domain.' )
  + ' AND    '
  + REPLACE( ISNULL(cm_target.filter_expr,'1=1'),'t.','t_domain.' )
  AS generated_SQL
FROM
  owl_object_property op,
  ontology o,
  object_property_map opm,
  class_map cm_source,
  class_map cm_target,
  db_table t_source,
  db_table t_target,
  db_database d,
  table_link tl1,
  db_table t_mid1,
  table_link t2,
  db_table t_mid2
WHERE
  op.ontology_id=o.ontology_id AND
  op.owl_object_property_id=opm.owl_object_property_id AND
  opm.source_class_map_id =cm_source.class_map_id AND
  opm.target_class_map_id =cm_target.class_map_id AND
  cm_source.db_table_id=t_source.db_table_id AND
  cm_target.db_table_id=t_target.db_table_id AND
  d.database_id=t_source.database_id AND
  tl1.table_link_id=opm.table_link_id AND
  t_mid1.db_table_id=tl1.mid_table_id
  tl2.table_link_id=tl1.next_table_link_id AND
  t_mid2.db_table_id=tl2.mid_table_id
```

A.7 RDB2OWL grammar in BNF notation

The grammar listed below is written in ANTLRWorks tool.

```
grammar RDB2OWL2;
options { backtrack=false; }

gMain  : classMap EOF;

classMap  : (defName '=')? tableExpr uriPattern? CDecoration*;
objectMap : tableExpr PDecoration*;
dataMap  : dataExpr PDecoration?;
ontologyDBExpr : (ontDBExprItem (';' ontDBExprItem)*)?;

ontDBExprItem : funDefPlus | 'CMap' '(' classMap ')' | dbSpec;
funDefPlus: functionDef | aggrFDef;
```

```
functionDef: fName '(' varList ')' '=' functionBody;
varList  : (variable (',' variable)*)? ;
functionBody: dataExpr;

aggrFDef : aggrUserFName '(' aggrArgList? ')' '=' functionBody;
aggrArgList: '@TExpr' '!' '@Col';

uriPattern: '{' 'uri' '=' '(' uriItem (',' uriItem)* ')' '}';
uriItem  : valueExpr;
tableExprPlain : simpleTableExpr | '(' tableExprExt ')';

tRefList : tRefItem (',' tRefItem)*;

tRefItem : tNavigItemE tRefItemL tExprTopSpec? ;
tRefItemL : (nLinkExpr tRefItemP)? ;
tRefItemP : (tNavigItem tRefItemL) | empty ;

dataExpr :(tableExprPlain )? '.' valueExprPlain ('^^' xsdRef)?;

tExprTopSpec: tTopFilter;
tableExpr : tRefList ( ';' tFilterExpr? (';' colDefList)?)?;


tNavigItemBase :simpleTableExpr refMark?|'(' tableExprExt ')'
refMark? ;
simpleTableExpr : tableRefExpr | ClassMapRef | namedRef ;

namedRef : '[[' defName ']]';
nLinkExpr : ('[' valueList ']')? ('->'|'=>') ('[' valueList ']')?;
valueList : valueExpr (',' valueExpr)*;

tNavigItem: tNavigItemBase (':' tNavigFilter)* ;
tNavigItemE: tNavigItem | empty ;
empty  : '.'?;

tNavigFilter: tFilterExpr | tTopFilter ;

tTopFilter: '{' ('first'|'top' INT ('percent')?) orderSpec? '}';
orderSpec : valueExpr ('asc'|'desc')?;

tableExprExt: tableExpr (uriPattern|keyPattern)?;
keyPattern: '{' 'key' '=' '(' keyItem (',' keyItem)* ')' '}';
keyItem  : valueExpr;

tFilterExpr: filterOrExpr ('or' filterOrExpr)*;
filterOrExpr: filterAndExpr ('and' filterAndExpr)*;
filterAndExpr: filterItem | 'not'? '(#' tFilterExpr '#)';

filterItem: unarybinaryFilterItem | constantFilterItem |
existsFilterItem | betweenFilterItem;
unarybinaryFilterItem: valueExpr ( 'is' 'not'? 'null' |
binaryFilterOp valueExpr);
constantFilterItem: 'true' | 'false';
binaryFilterOp : '=' | '<' | '>' | '<=' | '>=' | '<>' | 'like' | 'in'
;

existsFilterItem: 'exists' '(' tableExpr ')';
betweenFilterItem: valueExprPlain 'between' valueExprPlain 'and'
valueExprPlain;
```

```
colDefList: (colDef (',' colDef)*)?;
colDef  : VarName '=' valueExpr ;

simpleExpr: valueExprPlain | variable | functionCall  | prefixOp
simpleExpr
  | aggregateCall ;

valueExpr : simpleExpr (infixOp simpleExpr)*;
valueExprPlain : sqlExpr | colRef | INT | STRING | '(' valueExpr ')'
;
sqlExpr :caseTwoOptions | caseManyOptions;
caseTwoOptions: 'case' 'when' tFilterExpr 'then' valueExprPlain
('else'  valueExprPlain)? 'end';
caseManyOptions: 'case' valueExprPlain ('when' valueExprPlain 'then'
valueExprPlain)+ ('else' valueExprPlain)? 'end';

xsdRef : (XSD_TYPE_PREFIX)? VarName;

colRef  : colName | compoundColRef;
compoundColRef : simpleTableExpr '.' (colName | '(' colRef ')');
colName  : VarName;
colRefPlain: colName | '(' compoundColRef ')' ;

refMark  : VarName | ClassMapRef;
ClassMapRef: '<s>' | '<t>' | '<b>';
defName  : VarName;

dbOptionSpec :
 (
 'dbname=' VarName
 |
 'alias=' VarName
 |
 'schema=' STRING
 |
 'public_table_prefix=' STRING
 |
 'jdbc_driver=' STRING
 |
 'connection_string=' STRING
 |
 'aux=' ZEROONE
 |
 'default=' ZEROONE
 |
 'init_script=' STRING
 );
dbName  : VarName;
dbAlias  : VarName;
dbSpec  : 'DBRef' '(' dbOptionSpec (COMMA dbOptionSpec)* ')';
tableRefExpr : (dbAlias ':')? VarName;
variable : '@'VarName;
functionCall : fName '(' valueList ')';
fName  : VarName;
alias  : VarName;
infixOp  : '+' | '-' | '*' | '/' | 'div' | 'mod' ; //* continue ...
prefixOp : '-' ;
aggregateCall : aggrFName '(' dataExpr ')' | aggregateWrk;
```

```
aggrFName : 'min' | 'max' | 'avg' | 'count' | 'sum' | aggrUserFName;
aggrUserFName : '@' VarName;
aggregateWrk : '@aggregate' '(' tableExpr '!' valueExpr ',' valueExpr
','valueExpr ')';

CDecoration : '?' | '?Out' | '?In' | '!NoMap' | '!SubClean';
PDecoration: '?Domain' | '?Range' ;
VarName   :('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
COMMA :',';
ZEROONE :'0'|'1';
INT :'0'..'9'+;
COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    |   '/*' ( options {greedy=false;} : . )* '*/'
{$channel=HIDDEN;};
WS  :   ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;} ;
STRING :  '\'' ( ESC_SEQ | ~('\\'|'\'') )* '\'';
XSD_TYPE_PREFIX: 'xsd:';
fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ;

fragment
UNICODE_ESC :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
```

A.8 RDB2OWL semantic transformation code (Lquery, Lua)

A.8.1 util.lua

```
-- Utility functions used in other scripts
module(..., package.seeall)
function convertCollection2Array(collection)
 local theArray={}
 local nr=0

 table.sort(
  collection.objects,
  function(a,b)
   return a.id<b.id
  end
 )

 collection:each(
```

```
  function(element)
   nr=nr+1
   theArray[nr]=element
   return
  end
 )
 return theArray
end

function isTypeOf(object, className)
 return object:find("." .. className):is_not_empty()
end

function toSqlString(s)
 local retValue=""
 if (s=="null" or s==nil) then
  retValue="null"
 elseif (s=="") then
  retValue="''"
 else
  retValue="'" .. string.gsub(s, "'", "''") .. "'"
 end
 return retValue
end
```

A.8.2 rdb2owl_mm_libs.lua

```
-- RDB2OWL metamodel processing functions used in other scripts
module(..., package.seeall)
util = require("util")

function findMainTable(classMap)
 local firstRefItem=nil
 local firstNavigItem=nil
 local i=0
 local retValue=nil
 classMap:find("/tableExpression/refItem")
 :each(
  function(ri)
   i=i+1
   if (i==1) then
    firstRefItem=ri
   end
   return
  end
 )
 if (firstRefItem~=nil) then
  navigItems=firstRefItem:find("/navigItem")
  -- Sort navigItem objects by id
  table.sort(
   navigItems.objects,
   function(a,b)
    return a.id<b.id
   end
  )
  i=0
  navigItems
  :each(
```

```
   function(ni)
    i=i+1
    if (i==1) then
     firstNavigItem=ni
    end
    return
   end
  )
  if (firstNavigItem:find(".TableRefExpr"):is_not_empty() ) then
   retValue=firstNavigItem:find("/tableRef/ref")
  end
  if (firstNavigItem:find(".ExprItem"):is_not_empty() ) then
   retValue="ExprItem" --TODO
  end
  if (firstNavigItem:find(".ClassRef"):is_not_empty() ) then
   retValue="ClassRef"
   --find the only class map assigned to the referenced OWL class
   local i=0
   local referencesClassMap=nil
   firstNavigItem:find("/ref/classMap")
   :each(
    function(cm)
     i=i+1
     referencesClassMap=cm
     return
    end
   )
   if (i==1) then
    -- recursive call
    retValue=findMainTable(referencesClassMap)
   end
  end
  if (firstNavigItem:find(".DefVarRef"):is_not_empty() ) then
   -- recursive call
   retValue=findMainTable( firstNavigItem:find("/def/classMap") )
  end
 end

 return retValue
end

function getPrimaryKeyExpr(tab)
 local primaryKeyExpr=""
 local delimiter=""
 tab:find("/pKey/column"):each(
  function(column)
    primaryKeyExpr=primaryKeyExpr .. delimiter ..
column:attr("colName")
    delimiter=" || "
   return
  end
 )
 return primaryKeyExpr
end


--[[
 Converts expression objects into string text
 parameters:
```

```
 expr: object of type ValueExpression, BinaryExpression, etc which
should be transformet to String form
 tableAliasToAdd: new alias to add to table column expressions
 aliasChangeMap: alias change map is a Table with old values as keys
and new values as values, eg
  to change a1->b2, a2->b2 one would use:
  aliasChangeMap = {}
  aliasChangeMap["a1"]="b1"
  aliasChangeMap["a2"]="b2"
--]]
function toString(expr, tableAliasToAdd, aliasChangeMap)
 -- TODO: finish with all cases. Now are only covered only for mini-
univerity example
 local exprString=""
 local id=expr:id()
 local encloseInParentheses=false
 if (util.isTypeOf(expr,"ValueExpression") ) then
  if ( expr:find("Constant"):is_not_empty() ) then
   exprString=expr:attr("cValue")
  elseif ( expr:find("BinaryExpr"):is_not_empty() ) then
   local op= string.lower( expr:attr("op") )
--    encloseInParentheses= (op=="+" or op=="-" )
   local operands=util.convertCollection2Array(
expr:find("/valueExpression") )
   exprString= toString(operands[1], tableAliasToAdd, aliasChangeMap)
        .. op
        .. toString(operands[2], tableAliasToAdd, aliasChangeMap)


  elseif (util.isTypeOf(expr,"CompoundColRef") ) then
   local prefix=""
   if (expr:find("/colPrefix"):is_not_empty() and
expr:find("/colPrefix"):attr("text")~="" ) then
    prefix=expr:find("/colPrefix"):attr("text")
    if (aliasChangeMap~=nil and aliasChangeMap[prefix]~=nil) then
     prefix=aliasChangeMap[prefix]
    end
    prefix=prefix .. "."
   end
   if ( util.isTypeOf(expr:find("/columnRef"),"ColNameRef") ) then
    exprString=prefix .. expr:find("/columnRef"):attr("colName")
    if (tableAliasToAdd~=nil) then
     exprString= tableAliasToAdd .. "." .. exprString
    end
   end

  elseif ( expr:find("ColNameRef /tableColName"):is_not_empty() )
then
   local prefix=""
   if (expr:find("/colPrefix"):is_not_empty() and
expr:find("/colPrefix"):attr("text")~="" ) then
    prefix=expr:find("/colPrefix"):attr("text")
    if (aliasChangeMap~=nil and aliasChangeMap[prefix]~=nil) then
     prefix=aliasChangeMap[prefix]
    end
    prefix=prefix .. "."
   end
   exprString=prefix .. expr:attr("colName")
   if (tableAliasToAdd~=nil) then
    exprString= tableAliasToAdd .. "." .. exprString
```

```
   end


  elseif ( expr:find("/child"):is_not_empty() ) then
   encloseInParentheses=true
   exprString=toString( expr:find("/child"), tableAliasToAdd,
aliasChangeMap )
  end
 elseif (util.isTypeOf(expr,"BinaryFilterItem") ) then
  local operands=util.convertCollection2Array(
expr:find("/valueExpression") )
  exprString=toString(operands[1], tableAliasToAdd, aliasChangeMap)
   .. expr:attr("opName")
   .. toString(operands[2], tableAliasToAdd, aliasChangeMap)

 elseif (util.isTypeOf(expr,"AndOrFilter") ) then
  local operands=util.convertCollection2Array(
expr:find("/booleanExpr") )
  local a=toString(operands[1], tableAliasToAdd, aliasChangeMap)
  local b=toString(operands[2], tableAliasToAdd, aliasChangeMap)
  exprString = a.. " " ..expr:attr("cond") .. " "  .. b

 elseif (util.isTypeOf(expr,"UnaryFilterItem") ) then
  exprString=" " ..expr:attr("opName") .. " "
   .. toString(expr:find("/valueExpression"), tableAliasToAdd,
aliasChangeMap)
 elseif ( util.isTypeOf(expr,"NotFilter") ) then
  exprString=" " ..expr:attr("cond") .. " "
   .. toString(expr:find("/booleanExpr"), tableAliasToAdd,
aliasChangeMap)
 elseif (util.isTypeOf(expr,"BetweenExpr") ) then
  local operands=util.convertCollection2Array(
expr:find("/valueExpression") )
  exprString= toString(operands[1], tableAliasToAdd, aliasChangeMap)
   .. " BETWEEN "
   .. toString(operands[2], tableAliasToAdd, aliasChangeMap)
   .. " AND "
   .. toString(operands[3], tableAliasToAdd, aliasChangeMap)
 elseif (util.isTypeOf(expr,"ConstantFilter") ) then
  exprString=expr:attr("opName")
 end
 if (encloseInParentheses) then
  exprString = "(" .. exprString .. ")"
 end
 return exprString
end

function findFirstAndLastNavigItem(tableExpression)
 local firstNavigItem=nil
 local lastNavigItem=nil
 local refItems=tableExpression:find("/refItem")
 table.sort(
  refItems.objects,
  function(a,b)
   return a.id<b.id
  end
 )
 refItems:each(
  function(refItem)
   local navigItems=refItem:find("/navigItem")
```

```
   table.sort(
    navigItems.objects,
    function(a,b)
     return a.id<b.id
    end
   )
   navigItems:each(
    function(navigItem)
     if (firstNavigItem==nil) then
      firstNavigItem=navigItem
     end
     lastNavigItem=navigItem
     return
    end
   )
   return
  end
 )
 return firstNavigItem, lastNavigItem
end

function theOnlyClassMap(owlClass)
 local classMap=nil
 if (owlClass:find("/classMap"):size()==1) then
  classMap=owlClass:find("/classMap")
 end
 return classMap
end
```

A.8.3 LinkObjectPropertyMapsToClassMaps.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
 arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))


--[[
creates src and trg links from ObjectPropertyMap to ClassMap
instances.
src is determined as follows: denote p the OWL Object Property to
which the
ObjectPropertyMap is attachec. c- the only domain class of property
p.
src is the only class map ascribed to c.
trg is determined similarly (c is the only range class of property p)
--]]

colObjectPropertyMap=lQuery("ObjectPropertyMap")
colObjectPropertyMap
```

```
:each(
 function(c)
  local colSrcClassMap=nil
  local colTrgClassMap=nil
  --print( c:attr("localName") )
  c:find("/OWLObjectProperty/domain"):find("OWLClass")
   :each(
    function(c1)
     print("   domain: " .. c1:attr("localName"))
     classCount=c1:find("/classMap"):size()
     print("domain class count:" .. classCount)
     if classCount==1 then
      colSrcClassMap=c1:find("/classMap")
     end
     return
    end
   )
  c:find("/OWLObjectProperty/range"):find("OWLClass")
   :each(
    function(c1)
     print("   range: " .. c1:attr("localName"))
     classCount=c1:find("/classMap"):size()
     print("range class count:" .. classCount)
     if classCount==1 then
      colTrgClassMap=c1:find("/classMap")
     end
     return
    end
   )
  if (colSrcClassMap~=nill and colSrcClassMap:size()==1) then
   c:link("src", colSrcClassMap)
   print("link to src")
  end
  if (colTrgClassMap~=nill and colTrgClassMap:size()==1) then
   c:link("trg", colTrgClassMap)
   print("link to trg")
  end
  return
 end
)

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.4 linkDataTypePropertyMapsToClassMaps.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))
```

```
--[[
creates src from DataPropertyMap to ClassMap instances.
src is determined as follows: denote p the OWL Data Property to which
the
DataPropertyMap is attached. c- the only domain class of property p.
src is the only class map ascribed to c.
--]]


colObjectPropertyMap=lQuery("DataPropertyMap")
colObjectPropertyMap
 :each(
  function(c)
   local colSrcClassMap=nil
   c:find("/OWLDataProperty/domain"):find("OWLClass")
    :each(
     function(c1)
      print("   domain: " .. c1:attr("localName"))
      classCount=c1:find("/classMap"):size()
      print("domain class count:" .. classCount)
      if classCount==1 then
       colSrcClassMap=c1:find("/classMap")
      end
      return
     end
    )
   if (colSrcClassMap~=nill and colSrcClassMap:size()==1) then
    c:link("src", colSrcClassMap)
    print("link to src")
   end
   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.5 splitNamedRefToSubclasses.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
 arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))


--[[
Splittes ColNameref class instances into instances of its subclasses
DefColName and TableColName.
Decission which subclass to make is resolved by finding out if
colName property value is found
as property value of
```

```
  ColDef.colName in the TableExpression which is exprContext target
end (going from superclass ColumnRef)
If found then DefColName  instance is created otherwise- TableColName
instance.
--]]

lQuery("ColNameRef")
 :each(
  function(c)
   local colName=c:attr("colName")
   local refTarget=nill
   local subclassName="TableColName"
   local linkName="tableColName"
   print("   colName: " .. colName)
   local isFoundColDef=false
   c:find("/exprContext"):find("/colDef"):find("[colName=" .. colName
.. "]")
    :each(
     function(c1)
      if not isFoundColDef then
       isFoundColDef=true
       refTarget=c1
       subclassName="DefColName"
       linkName="defColName"
       print("        is found as DefColName")
      end
      return
     end
    )


   local newSubClass=lQuery.create(subclassName)
   c:link(linkName, newSubClass)

   if subclassName=="DefColName" and refTarget~=nil then
    newSubClass:link("ref", refTarget)
   end

   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.6 linkColPrefixToNavigItem.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
 arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))
```

```
--[[
Creates ref links from ColPrefix to NavigItem class instances.
For example, in table expression
Person p, Company c; p.company_id=p.company_id
ColPrefix p (in p.company) refers to "Person p" navigation item which
is TableRefExpr with alias=p in this case.
Alghoritm to find ref link of ColPrefix instance cp
1) Determine context c of ColPrefix cp: ColPrefix --> ColumnRef -->
(exprContext) TableExpression
2)
 for each RefItem r of c
  for each NavigItem n of c
   if n is of type ClassMapRef and n.mark=cp.text then n is ref link
target
   if n is of type ExprItem and n.alias=cp.text then n is ref link
target
   if n is of type NamedRef and n.refName=cp.text then n is ref link
target
   if n is of type TableRefExpr and n.alias=cp.text then n is ref
link target
--]]

lQuery("ColPrefix")
 :each(
  function(cp)
   local prefixText=cp:attr("text")
   local context=cp:find("/columnRef/exprContext")
   local refItem= context:find("/refItem")
   local ref=-1
   local found=false
   refItem:find("/navigItem"):find("ClassMapRef")
    :each(
     function(ni)
      if ni:attr("mark")==prefixText then
       ref=ni
       found=true
      end
      return
     end
    )
   if not found then
    refItem:find("/navigItem"):find("AggrBaseRef")
     :each(
      function(ni)
       if ni:attr("mark")==prefixText then
        ref=ni
        found=true
       end
       return
      end
     )
   end
   if not found then
    refItem:find("/navigItem"):find("ExprItem")
     :each(
      function(ni)
       if ni:attr("alias")==prefixText then
        ref=ni
        found=true
```

```
        end
        return
       end
      )
   end
   if not found then
    refItem:find("/navigItem"):find("NamedRef")
     :each(
      function(ni)
       if ni:attr("refName")==prefixText then
        ref=ni
        found=true
       end
       return
      end
     )
   end
   if not found then
    refItem:find("/navigItem"):find("TableRefExpr")
     :each(
      function(ni)
       if ni:attr("alias")==prefixText then
        ref=ni
        found=true
       end
       return
      end
     )
   end

   if found then
    cp:link("ref", ref)
   end
   return
  end
)



print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.7 changeEmptyItemsToClassMapRefs.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))


--[[
```

```
changes empty ClassMapRef instances (mark="empty") to non empty
ClassMapRef instances (mark="<s>" or mark="<t>"):
if the first NavigItem of navigList (as RefItem -->{ordered}
NavigItem)
is empty item then change it to ClassMapRef istance with mark="<s>";
if the last NavigItem of navigList is EmptyItem then change it to
ClassMapRef istance with mark="<t>"
It means insertion of missing <s> or <t> , e.g.  "=>Registration->"
changed to "<s>=>Registration-><t>"
--]]


colRefItem=lQuery("RefItem")
colRefItem
 :each(
  function(c)
   local objectPropertyMap=
c:find("/tableExpression/objectPropertyMap")
   local belongsToObjectPropertyMap = objectPropertyMap:size()>0
   local dataPropertyMap=
c:find("/tableExpression/dataExpression/dataPropertyMap")
   local belongsToDataPropertyMap = dataPropertyMap:size()>0
   local colNavigItem=c:find("/navigItem")
   local navigItemCount= colNavigItem:size()

   table.sort(
    colNavigItem.objects,
    function(a,b)
     return a.id<b.id
    end
   )

   local nr=0
   print("---------------------------------")
   colNavigItem
    :each(
     function(c1)
      c1:log()
      nr=nr+1
      isEmptyItem = c1:find("ClassMapRef[mark=empty]"):size()==1
      if nr==1 then --the first navigItem
       if isEmptyItem then
        print("                      this first is EmptyItem" )
        if belongsToObjectPropertyMap
         or belongsToDataPropertyMap then
         c1:attr("mark", "<s>")
        end
       end
      end
      if nr==navigItemCount then --the last navigItem
       if isEmptyItem then
        print("                      this last is EmptyItem" )
        if belongsToObjectPropertyMap then
         c1:attr("mark", "<t>")
        end
       end
      end
      return
     end
    )
```

```
   return
  end
 )


print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.8 linkClassMapRef2ClassMap.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
 arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))


--[[
Make links ClassMapRef -->ref ClassMap
--]]


colRefItem=lQuery("ClassMapRef")
colRefItem
 :each(
  function(c)
   local objectPropertyMap=
     c:find("/refItem/tableExpression/objectPropertyMap")
   local belongsToObjectPropertyMap = objectPropertyMap:size()>0
   local dataPropertyMap=
    c:find("/refItem/tableExpression/dataExpression/dataPropertyMap")
   local belongsToDataPropertyMap = dataPropertyMap:size()>0
   local mark=c:attr("mark")
   if mark=="<s>" then
    if belongsToObjectPropertyMap then
     print("link to /src")
     src=objectPropertyMap:find("/src")
     print("src:size()=", src:size())
     c:link("ref", objectPropertyMap:find("/src") )
    end
    if belongsToDataPropertyMap then
     print("link to /src.")
     c:link("ref", dataPropertyMap:find("/src") )
    end
   end
   if mark=="<t>" then
    if belongsToObjectPropertyMap then
     print("link to /trg")
     c:link("ref", objectPropertyMap:find("/trg") )
    end
   end
```

```
   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.9 linkXSDRef2XSDDatatype.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
 arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))

--[[
links XSDRef instances to corresponding XSDDatatype instance based on
typeName property value equality
--]]

lQuery("XSDRef")
 :each(
  function(c)
   local typeName=c:attr("typeName")
   lQuery("XSDDatatype[typeName=" .. typeName .."]")
    :each(
     function(c1)
      c:link("ref", c1)
      return
     end
    )
   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.10 splitColNameRef2Subclasses.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
```

```
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))


--[[
Splittes ColNameref class instances into instances of its subclasses
DefColName and TableColName.
Decission which subclass to make is resolved by finding out if
colName property value is found
as property value of
  ColDef.colName in the TableExpression which is exprContext target
end (going from superclass ColumnRef)
If found then DefColName  instance is created otherwise- TableColName
instance.
--]]

lQuery("ColNameRef")
 :each(
  function(c)
   local colName=c:attr("colName")
   local refTarget=nill
   local subclassName="TableColName"
   local linkName="tableColName"
   print("   colName: " .. colName)
   local isFoundColDef=false
   c:find("/exprContext"):find("/colDef"):find("[colName=" .. colName
.. "]")
    :each(
     function(c1)
      if not isFoundColDef then
       isFoundColDef=true
       refTarget=c1
       subclassName="DefColName"
       linkName="defColName"
       print("        is found as DefColName")
      end
      return
     end
    )


   local newSubClass=lQuery.create(subclassName)
   c:link(linkName, newSubClass)

   if subclassName=="DefColName" and refTarget~=nil then
    newSubClass:link("ref", refTarget)
   end

   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.11 fillExpliciteNavigationColumns.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
 arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))


--[[
Inserts explicit column informartion into navigation links:
Table1->Table2 changes to Table1[colA]->[colB]Table2
or
Table1->Table2 changes to Table1[colA1, colA2, ..., colAn]->[colB1,
colB2, ..., colBn]Table2
--]]

function findPkColumns(tableName)
  local colList={}
  local i=0
  lQuery("Table[tName=" .. string.upper(tableName) ..
"]"):find("/pKey/column")
 :each(
  function(c)
   i=i+1
   colList[i]=c:attr("colName")
   return
  end
 )
  return colList
end

function findFkColumns(sourceTableName, targetTableColumn)
  local colList={}
  local i=0
  -- find FKey that links source table with target table
  local fKey=lQuery("Table[tName=" .. string.upper(sourceTableName)
.. "]")
 :find("/column/fKey[/target@tName=" ..
string.upper(targetTableColumn) .."]")
 fKey:find("/column")
 :each(
  function(c)
   i=i+1
   colList[i]=c:attr("colName")
   return
  end
 )
  return colList
end

function findTable(navigItem)
--[[
 to determine the table for the NavigationItem we should try
different cases:
```

```
 1: NavigItem as ClassMapRef
  1.1 If mark=<s> then
   objectPropertyMap->(src)ClassMap->tableExpression->
    refItem->navigItem->tableRef.tName
  1.2 If mark=<t> then
   objectPropertyMap -> (trg)ClassMap -> tableExpression
    -> refItem -> navigItem -> tableRef.tName
 2: NavigItem as TableRefExpr -> tableRef.tName
 3: NavigItem as ClassRef -> (ref)OWLClass
    ->classMap->tableExpression->refItem->navigItem->tableRef.tName
 3: NavigItem as DefVarRef -> (def)Reference ->classMap
    ->tableExpression->refItem->navigItem->tableRef.tName
--]]
 local tableRefExpr=navigItem:find(".TableRefExpr")
 local classMapRef=navigItem:find(".ClassMapRef")
 local classRef=navigItem:find(".ClassRef")
 local defVarRef=navigItem:find(".DefVarRef")
 local dbTable
 if classMapRef:size()==1 then -- navigItem is class map ref
  if classMapRef:attr("mark")=="<s>" then

 dbTable=objectPropertyMap:find("/src/tableExpression/refItem/navigIt
em")
    :find(".TableRefExpr/tableRef"):attr("tName")
  end
  if classMapRef:attr("mark")=="<t>" then

 dbTable=objectPropertyMap:find("/trg/tableExpression/refItem/navigIt
em")
    :find(".TableRefExpr/tableRef"):attr("tName")
  end
 end
 if tableRefExpr:size()==1 then
  dbTable=tableRefExpr:find("/tableRef"):attr("tName")
 end
 if classRef:size()==1 then

 dbTable=classRef:find("/ref/classMap/tableExpression/refItem/navigIt
em")
   :find(".TableRefExpr/tableRef"):attr("tName")
 end
 if defVarRef:size()==1 then

 dbTable=defVarRef:find("/def/classMap/tableExpression/refItem/navigI
tem")
   :find(".TableRefExpr/tableRef"):attr("tName")
 end
 return dbTable
end



lQuery("NavigLink")
 :each(
  function(c)
   local symbol=c:attr("symbol")
   local sourceTable=nil
   local targetTable=nil
   local tableExpression=c:find("/left/refItem/tableExpression")
```

```
 objectPropertyMap=c:find("/left/refItem/tableExpression/objectProper
tyMap")

 objectPropertyName=objectPropertyMap:find("/OWLObjectProperty"):attr
("localName")
   sourceClassMap=objectPropertyMap:find("/src")
   sourceClass=sourceClassMap:find("/OWLClass")
   sourceClassName=sourceClass:attr("localName")
   if sourceClassName==nil then
    sourceClassName="noClass"
   end
   -- if left column list is missing, eg: TABLE1 -> [target column
list] TABLE2
   sourceTable=findTable(c:find("/left"))
   targetTable=findTable(c:find("/right"))
   local sourceColList
   local targetColList
   if (symbol=="->") then
    sourceColList=findFkColumns(sourceTable, targetTable)
    targetColList=findPkColumns(targetTable)
   else
    sourceColList=findPkColumns(sourceTable)
    targetColList=findFkColumns(targetTable, sourceTable)
   end

   if c:find("/leftC"):size() == 0  then
    local newValueList=lQuery.create("ValueList")
    c:link("leftC", newValueList)
    for i,col in ipairs(sourceColList) do
     local newColNameRef=lQuery.create("ColNameRef")
     newColNameRef:link("exprContext", tableExpression)
     local newTableColName=lQuery.create("TableColName")
     newColNameRef:link("tableColName", newTableColName)
     newColNameRef:attr("colName", col)
     newValueList:link("valueExpression", newColNameRef)
    end
   end

   -- if right column list is missing, eg: TABLE1[target column
list]->TABLE2
   if c:find("/rightC"):size()==0 then
    local newValueList=lQuery.create("ValueList")
    c:link("rightC", newValueList)
    for i,col in ipairs(targetColList) do
     local newColNameRef=lQuery.create("ColNameRef")
     newColNameRef:link("exprContext", tableExpression)
     local newTableColName=lQuery.create("TableColName")
     newColNameRef:link("tableColName", newTableColName)
     newColNameRef:attr("colName", col)
     newValueList:link("valueExpression", newColNameRef)
    end
   end

   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
```

```
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.12 linkObjectPropertyMapsToClassMapsByNamedRefs.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
mm_libs = require("rdb2owl_mm_libs")
if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))

--[[
creates src and trg links from ObjectPropertyMap to ClassMap
instances when
they are not created by default algorithm: (unique class map of
domain/range class of the property)
src class map of object property map is determined as follows:
  1) let x be the first navigItem of the first refItem
  2) if x is of type 'NamedRef' then src class map is
referencedClassMap(x)
trg class map of object property map is determined as follows:
  1) let x be the last navigItem of the last refItem
  2) if x is of type 'NamedRef' then trg class map is
referencedClassMap(x)

referencedClassMap(x) is determined as follows:
  1) if x is of type 'ClassRef' and c is class that is linked to x by
ref link
       then return the only class map that is ascribed to class c
  2) if x of of type 'DefVarRef' then return class map linked to x:
      x -->(def) --> classMap
 --]]

-- For example, this script creates src and trg links for two object
property maps
--  for properety PersonID (mimi university example):
--  2 class maps of PersonID class:
--    classMap1: S=Student {uri=('PersonID', IDCode)}
--    classMap2: T=Teacher {uri=('PersonID', IDCode)}
--  2 property maps of object property personID:
--   propMap1: [[ Student]][student_id]->[[S]]
--   propMap2: [[Teacher]][teacher_id]->[[T]]
--  In this example:
--    propMap1 -->(src) The only class map of Student class
--    propMap1 -->(trg) classMap1
--    propMap2 -->(src) The only class map of Teacher class
--    propMap1 -->(trg) classMap2



tableExpressions=lQuery("ObjectPropertyMap/tableExpression")
tableExpressions
```

```
:each(
 function(tableExpression)
  objectPropertyMap=tableExpression:find("/objectPropertyMap")
  local isNotSrc=objectPropertyMap:find("/src"):is_empty()
  local isNotTrg=objectPropertyMap:find("/trg"):is_empty()
  local sourceClassMap=nil
  local targetClassMap=nil
  if (isNotSrc or isNotTrg) then
   local firstNavigItem, lastNavigItem =
mm_libs.findFirstAndLastNavigItem(tableExpression)

    local start, finish
    if (firstNavigItem:find(".ClassRef"):is_not_empty() ) then

 sourceClassMap=mm_libs.theOnlyClassMap(firstNavigItem:find("/ref"))
    elseif (firstNavigItem:find(".DefVarRef"):is_not_empty() ) then
     sourceClassMap= firstNavigItem:find("/def/classMap")
    end
    if (lastNavigItem:find(".ClassRef"):is_not_empty() ) then

 targetClassMap=mm_libs.theOnlyClassMap(lastNavigItem:find("/ref"))
    elseif (lastNavigItem:find(".DefVarRef"):is_not_empty() ) then
     targetClassMap= lastNavigItem:find("/def/classMap")
    end
    if (isNotSrc and sourceClassMap~=nil) then
     objectPropertyMap:link("src", sourceClassMap)
     print("create src link from ObjectPropertyMap:" ..
objectPropertyMap:id() .. " to classMap:" .. sourceClassMap:id())
    end
    if (isNotTrg and targetClassMap~=nil) then
     objectPropertyMap:link("trg", targetClassMap)
     print("create trg link from ObjectPropertyMap:" ..
objectPropertyMap:id() .. " to classMap:" .. targetClassMap:id())
    end
   end

   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.13 linkDataPropertyMapsToClassMapsByNamedRefs.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
mm_libs = require("rdb2owl_mm_libs")
if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))
```

```
--[[
creates src links from DataPropertyMap to ClassMap instances when
they are not created by default algorithm: (unique class map of
domain class of the property)
src class map of object property map is determined as follows:
  1) let x be the first navigItem of the first refItem
  2) if x is of type 'NamedRef' then src class map is
referencedClassMap(x)

referencedClassMap(x) is determined as follows:
  1) if x is of type 'ClassRef' and c is class that is linked to x by
ref link
       then return the only class map that is ascribed to class c
  2) if x of of type 'DefVarRef' then return class map linked to x:
      x -->(def) --> classMap
 --]]

tableExpressions=lQuery("DataPropertyMap /dataExpression
/tableExpression")
tableExpressions
 :each(
  function(tableExpression)
   dataPropertyMap=tableExpression:find("/dataExpression
/dataPropertyMap")
   local isNotSrc=dataPropertyMap:find("/src"):is_empty()
   local sourceClassMap=nil
   if (isNotSrc ) then
    local firstNavigItem, temp =
mm_libs.findFirstAndLastNavigItem(tableExpression)

    local start, finish
    if (firstNavigItem:find(".ClassRef"):is_not_empty() ) then

 sourceClassMap=mm_libs.theOnlyClassMap(firstNavigItem:find("/ref"))
    elseif (firstNavigItem:find(".DefVarRef"):is_not_empty() ) then
     sourceClassMap= firstNavigItem:find("/def/classMap")
    end
    if (isNotSrc and sourceClassMap~=nil) then
     dataPropertyMap:link("src", sourceClassMap)
     print("create src link from DataPropertyMap:" ..
dataPropertyMap:id() .. " to classMap:" .. sourceClassMap:id())
    end
   end

   return
  end
 )



print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.14 createMissingLink_exprContext.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))

--[[
creates exprContext links:
ColumnRef -->(exprContext) TableExpression
--]]

lQuery("ColumnRef:not(/exprContext)")
:each(
 function(columnRef)
  --columnRef:log()
  evalContext=columnRef:find("/evalContext")
  if ( evalContext:is_not_empty() ) then
   local tableExpression=
evalContext:find("/dataPropertyMap/src/tableExpression")
   if ( tableExpression:is_not_empty() ) then
    columnRef:link("exprContext", tableExpression)
   end
   print("create link ColumnRef:" .. columnRef:id() .. " --
>(exprContext) DataExpression:" .. evalContext:id() )
  end
  return
 end
)

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.15 create_refLinks2TableAndColumn.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))
local dbAlias=""

--[[
1) Create ref link from TableRef to Table
2) Create ref link from TableColName to Column
--]]
```

```
--First fill empty TableRef.dbAlias with
--alias of default database (DBRef.isDefault=true)
defaultDBRef=lQuery("DBRef[isDefaultDB=true]")
if defaultDBRef:size()>1 then
 print("ERROR: more than 1 default Database reference specified in
ontology annotation " )
 return
end

if defaultDBRef:size()==1 then
 local defaultDBAnnotation = defaultDBRef:attr("dbAlias")
 print("defaultDBAnnotation=" .. defaultDBAnnotation)
 lQuery("TableRef")
 :each(
  function(tableRef)
   print( tableRef:attr("dbAlias") )
   dbAlias=tableRef:attr("dbAlias")
   if dbAlias=="" or dbAlias==nil then
    dbAlias=defaultDBAnnotation
    tableRef:attr("dbAlias", defaultDBAnnotation)
   end
   --create ref link from TableRef to Table
   local database=lQuery("DBRef[dbAlias=" .. dbAlias .."]/refDb")
   local tableName=tableRef:attr("tName")
   local colTable=database:find("/table[tName="
     .. string.upper(tableName) .."]")
   print("tableName=" .. tableName .. ",  colTable:size="
     .. colTable:size() )
   tableRef:link("ref", colTable)
   return
  end
 )
end

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```

A.8.16 insertDefaultURIPatterns.lua

```
log = print
require "lua_mii_rep"
lQuery = require "lQuery"
mm_libs = require("rdb2owl_mm_libs")

if (arg[1]==nil) then
  arg[1]="C:\\mii_repozitorijs\\rdb2owl_mm"
end
local repo_path = arg[1]
print("open repo at " .. repo_path,
assert(lua_mii_rep.Connect(repo_path)))

nr=0
-- class maps without explicite URI pattern:
lQuery("ClassMap:not(/URIPattern)")
 :each(
  function(classMap)
```

```
   nr=nr+1
   local tableExpression=classMap:find("/tableExpression")
   local mainTable=mm_libs.findMainTable(classMap)
   local tableName=mainTable:attr("tName")
   if (mainTable==nil) then
    mainTable="no main table"
   end
   print(nr .. ". " .. classMap:id()
   .. ", OWLClass=" .. classMap:find("/OWLClass"):attr("localName")
   .. ", table=" .. tableName
   )
   local URIPattern=lQuery.create("URIPattern")
   classMap:link("URIPattern", URIPattern)
   local URIItem=lQuery.create("URIItem")
   URIPattern:link("URIItem", URIItem)
   local valueExpr=lQuery.create("Constant")
   valueExpr:attr("cValue", "'" ..tableName .. "'")
   URIItem:link("valueExpression", valueExpr)
   -- add all PrimaryKey columns to the URIPattern
   mainTable:find("/pKey/column")
   :each(
    function(column)
     local colName=column:attr("colName")
     local colNameRef=lQuery.create("ColNameRef")
     colNameRef:attr("colName", colName)
     local tableColName=lQuery.create("TableColName")
     colNameRef:link("tableColName", tableColName)
     colNameRef:link("exprContext", tableExpression)

     URIItem=lQuery.create("URIItem")
     URIPattern:link("URIItem", URIItem)
     URIItem:link("valueExpression", colNameRef)

     return
    end
   )

   return
  end
 )

print("save repo at " .. repo_path, assert(lua_mii_rep.Save()))
print("close repo at " .. repo_path,
assert(lua_mii_rep.Disconnect()))
```