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, 2011

EIROPAS SOCIĀLAIS FONDS

EIROPAS SAVIENĪBA

IEGULDĪJUMS TAVĀ NĀKOTNĒ

LATVIJAS UNIVERSITĀTE
ANNO 1919

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

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 the World Wide Web scale, as well as on 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.

We start by reviewing the basic ideas behind the semantic web and semantic technologies. Then we outline for task of semantic re-engineering of relational databases and discuss the challenges and benefits for relational data mapping into the semantic technology format. We conclude the introduction by briefly presenting the existing solutions in the area of relational data to semantic technologies mappings, sketching the need for our solution and its basic characteristics, as well as by providing the overall structure of the theses.

1.1 Semantic Web and Semantic Technologies

Semantic Web is group of methods, technologies and tools to make the huge information in the World Wide Web to be processable by machines and semantic understandable also by machines. The main technologies for this purpose are RDF[1], RDFS[2], OWL[3] among others.

RDF is language to describe information about resourses in the World Wide Web. The information is described as a list of statements each statement being Subject-Predicate-Object triple. Subject- resource about 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 some statement can be resource- another statement therefore triple set (described in RDF language) can be viewed as oriented graph with nodes denoting Subjects and Objects of the triple and arcs standing for Predicates directed from Subject to Object.

As an example, consider a simple statement "Population of Riga a capital of Latvia is 706413". Below is shown RDF graph holding this information using fictious vocabulary http://geography obtained from RDF Validation Service [5]:

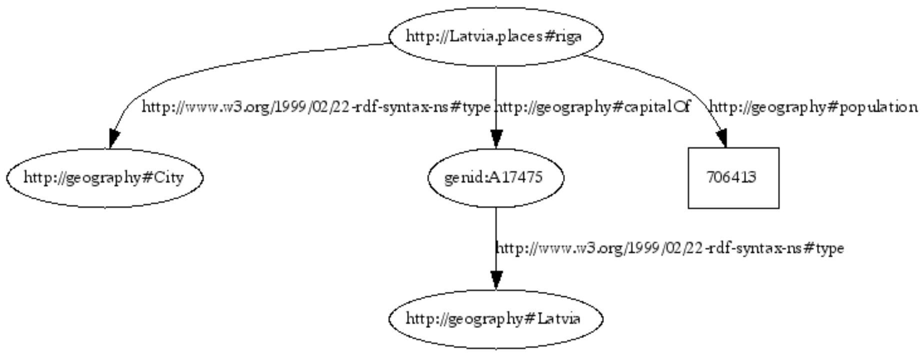

http://Latvia.places#riga
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://geography#capitalOf
http://geography#population
http://geography#City
genid:A17475
706413
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://geography#Latvia

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

Blank nodes (genid:A17475 in the example) are referencable only from within 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 can be 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)[2] is language that allows semantical extension of RDF by introducing grouping of resources and relations between them. This is done by classes and properties. RDF document can use RDFS vocabularies to specify types of resources and properties (eg, *http://geography#City*, http://geography#capitalOf). Web Ontology Language OWL[3] 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).

In short Semantic Web technologies RDF, RDFS, OWL and others enables information to be coded in globally accessable, machine processable way and also more conforming to semantic of domain of discourse and logic than some other information coding systems (eg, relational tables).

With the advent of Semantic Web technologies and the need to enable those to access the massive amount of data that are existing in the form of relational databases (RDB) both in public domain and proprietary in companies and organizations, the need of bridging the RDB and semantic RDF/OWL data formats has become apparent and has been widely studied.

1.2 RDB-to-RDF/OWL mapping solutions

Some of the most notable approaches dealing with RDB to RDF/OWL data mapping are Relational.OWL[6, 7], R2O [8], D2RQ [9], Virtuoso RDF Views [10] and DartGrid [11], Ultrawrap [12], Triplify[13]. There is W3C RDB2RDF Working Group [14] related to standardization of RDB to RDF mappings, as well as a related published survey of mapping RDBs to RDF [15].

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 be used when data that resides in relational databases are to be made available in the global scope of the World Wide Web in form of RDF triples (This is case for Open Government initiative TODO: link []) but resources are short to make appropriate restructuring of data. When raw data are exposed in RDF form any one is free to write appropriate processing code and thus open development is promoted.

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 RDF dataset consist of triples that are instances of ontology that is preexistant independently from database. We say that the source database is mapped to the target ontology (thought as conceptual model for the database).

Relational.OWL is a direct mapping platform that enables transformation of relational schema and data to OWL and RDF coding and therefore to be accessable from SPARQL endpoint. It gives base for mapping solutions between OWL ontologies, eg. expressing mappings by SPARQL. R2O platform consists of declarative mapping language between relational DB and OWL ontology and of tools (Mapster) to process these mappings. RO2 language requires rather leanghty writing if done by hand, but some help for this is in user interface of Mapster. R2O is suitable when mapping code is automatically generated. D2RQ is platform constisting with 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 also has language to express mapping between RDB and OWL ontologies and also has tools to process SPARQL accessing reational data on the fly through these mappings. Virtuoso RDF RDB-to-OWL mapping language is more complicated than R2O and D2RQ thas giving option to specify more execution details (eg, functions for URI patterns). Ultrawrap and Triplify platforms use SQL elaboration to enable triple generation from RDB data. Week point is that SQL for triple generation need to be written manually. Triplify has olso small plugin (PHP) which added to web application root enables triplifying of relational data when triplifying SQLs and configuration (connections) are provided.

There is upcoming W3C standard R2RML [16] for RDB-to-RDF mapping specification, as well as a standard for direct (technical) mapping [17] of RDBs to RDF format. In the case of the latter approach 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 [6, 7] approach, or by means of some RDF-to-RDF mapping language such as R2R [18], or some model transformation language (see e.g. [19] for an example approach).

The initial RDB-to-RDF mapping definition by means of hand-coded SQL statements as outlined in [20] 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.

1.3 Semantic Re-engineering of Relational Databases

The development of Semantic Web has been for more than a decade. As a result there are many semantic tools for semantic data management but few data in semantic form (eg., RDF triple stores) because most of them still resides in relational databases, closed for public access. Therefore the need of high importance is 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 [21] 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 [21] 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 ar still to come. To publish relational data as RDF corresponding to target ontology can be done by using some mapping specification techniques.

The possibility to define RDB-to-RDF/OWL mappings efficiently has emerged as an issue of primary importance also in Semantic Latvia approach [22] and its application to the practical semantic re-engineering of medical domain data in Latvia [23, 24]. This approach proposes creating ontology (ontologies) for data that are available in a specific domain (e.g., government data, or medical data), using visual graphical notation offered by OWLGrEd [25,26] or UML/OWL profile [27], 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 [23,28].

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 [29] 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 [30] 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. So from data.gov.uk raw data can be accessable as RDF. People can build applications over those raw data or write code to transform them into meaningfull ontologies. In this scenarious the burden of re-engeneering of relational databases is left on open development community. This is in accordance to what Tim Berner-Lee said on talk in TED2009 conference [31] "Raw Data Now". Re-engeneering thus is spluit into two phases: in the first one the data publisher technically publises the raw data and in the other phase the open community makes data thansforation into various domain ontologies. In order to do these tasks more effectively methods and tools for semantic re-engeneering are needed.

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 is extensible beyond those. Our solution contains a mapping language and implementation framework briefly described below.

We prepose a high level, human readable and machine processable declarative RDB-to-RDF/OWL mapping specification language RDB2OWL that is based on re-using the target ontology structure as a backbone where mapping expressions can be written in 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 attachement 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.

The RDB2OWL execution environment is technically based on a designated relational database schema to store the mapping information, and the triple generation is done by two-phase SQL processing. 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 [23], [24] 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, the Latvian Medical registry ontology has been successfully annotated with RDB2OWL annotations. 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 mapping expressions of RDB2OWL mapping language grammar;
- MOF-style RDB2OWL mapping metamodel;
- mapping pattern observations from real life examples that is supported by RDB2OWL mapping language;
- execution architecture with mappings stored in relational database and two phase SQL execution.

2 RDB-to-RDF/OWL mapping task

2.1 Need for RDB-to-RDF/OWL mapping solutions

The recent years are characterized by increasing use of semantic technologies both in 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 [47], OWL 2.0 [4] and many others. The number and performance of tools for semantic content management has grown and continue to grow. But majority of data still continue to reside in relational databases. Some reasons: relational databases are efficienct in terms of processing time and data volume, they have precise definition, many SQL based tools and application development environments, a lot of existing applications use data in relational databases.

In this situation the need of high importance is 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 research and technology development has been aimed at bridging relational databases (RDB) to RDF/OWL by mapping languages and techniques, starting with paper "Relational Databases on the Semantic Web" by T.Berners-Lee [32] back in 1998 and continuing with a number of successful approaches including R_2O [8], D2RQ [9], Virtuoso RDF Views [10], DartGrid [33] as well as on UltraWrap [12],Triplify [13] and Spyder [34].

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 [35] can be mentioned, 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 long an issue within MOF-centered [36] model transformation community. A model transformation languages such as MOF QVT [37], MOLA [38] or AGG [39] (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 [19].

Human readable RDB-to-RDF/OWL mapping language can be used as a documentation means where database tables and columns are mapped to conceptual classes and attributes. This approach has notable advantages over textual and graphical documentation means. Possible advantages are: documentation is human readable and formal at the same time, precise, machine processable (eg, for validation, reporting, etc) and tracable in both directions.

2.2 RDB and semantic format comparison

There are substantial differences comparing RDB vs. RDFS/OWL data models listed below.

- Naming of entities and relations in relational databases are technically oriented but in RDF/OWL conceptual names are used from specifid domain;
- relational databases are not aware of subclass relation but this is one of most basic relation used in ontologies.
- It is not possible to have n:n relation between two database tables (third table needs to be used) but RDFS/OWL ontologies has not such problem.
- Relations (foreign key) between database tables does 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 cnstraints on domain or range classe. Databases can easily express 0..1 cardinality by using NOT NULL constraint The other constraints can be dealt by trigger programming but then qualifier information is burried in code and not easily seen in model.

2.3 Working examples

In this section we introduce three examples that will be used 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

To better explain various approaches for bridging relational databases and OWL/RDFS ontologies, a simple example database reflecting a miniature study registration system and the related OWL ontology will be used. It is taken from [20]. Below in fig.2. and fig.3. are shown sample database schema and a 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; OWL class *PersonID* 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 *Registration* table.

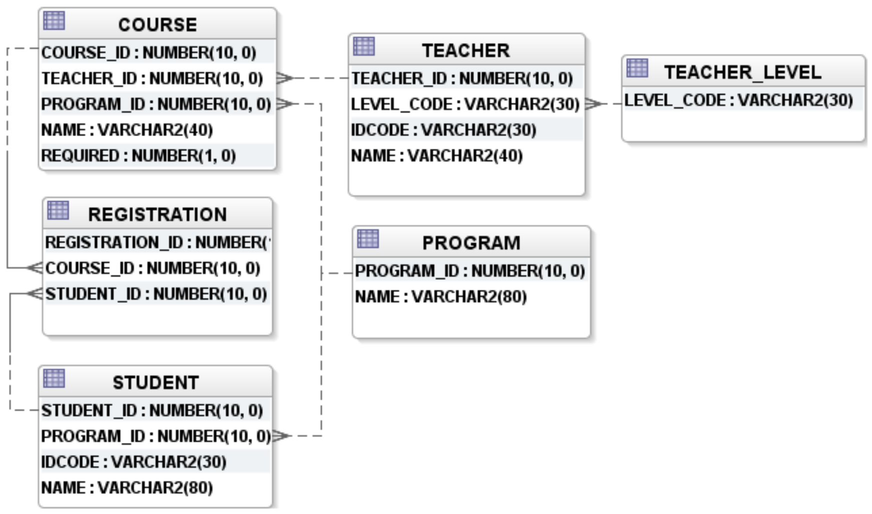

COURSE
COURSE_ID : NUMBER(10, 0)
TEACHER_ID : NUMBER(10, 0)
PROGRAM_ID : NUMBER(10, 0)
NAME : VARCHAR2(40)
REQUIRED : NUMBER(1, 0)
TEACHER
TEACHER_ID : NUMBER(10, 0)
LEVEL_CODE : VARCHAR2(30)
IDCODE : VARCHAR2(30)
NAME : VARCHAR2(40)
TEACHER_LEVEL
LEVEL_CODE : VARCHAR2(30)
REGISTRATION
REGISTRATION_ID : NUMBER(
COURSE_ID : NUMBER(10, 0)
STUDENT_ID : NUMBER(10, 0)
PROGRAM
PROGRAM_ID : NUMBER(10, 0)
NAME : VARCHAR2(80)
STUDENT
STUDENT_ID : NUMBER(10, 0)
PROGRAM_ID : NUMBER(10, 0)
IDCODE : VARCHAR2(30)
NAME : VARCHAR2(80)

Fig. 2. Mini-university relational database schema

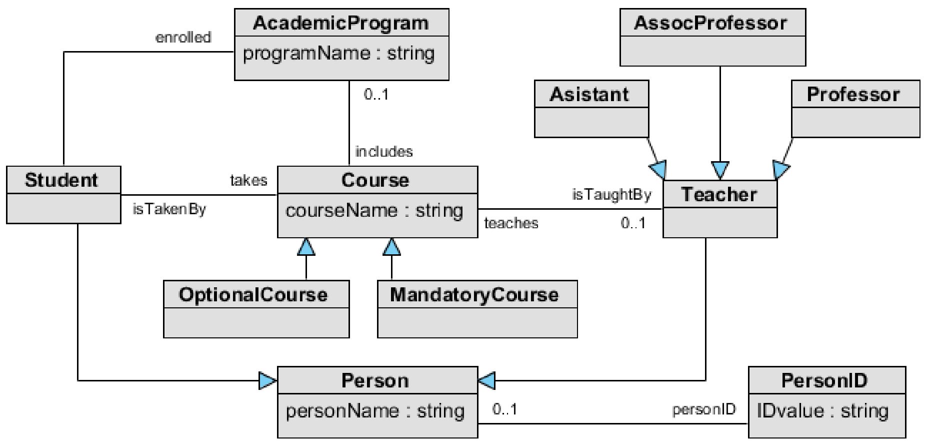


Fig. 3. Mini-university ontology

For example, the classes *Student* and *Course* in this sample ontology have corresponding tables *student* and *course* in the sample database. To get instance data for OWL object property *takes* the table link path is needed: tables *student* and *registration* joined on *student_id* and *registration* and *course* joined on *course_id*.

Class *personID* instances are populated from *student* and *teacher* tables (*idcode* column).

Classes *Asistant*, *AssocProfessor* and *Professor* all get instance data from common table *TEACHER* 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 following tables we show the actual data tables of our sample database. This specific data set will be used as an example.

Table 1. Table *program* data

program_id	**name**
1	Computer Science
2	Computer Engeneering

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 data`base with far linking tables with schema and data as below.

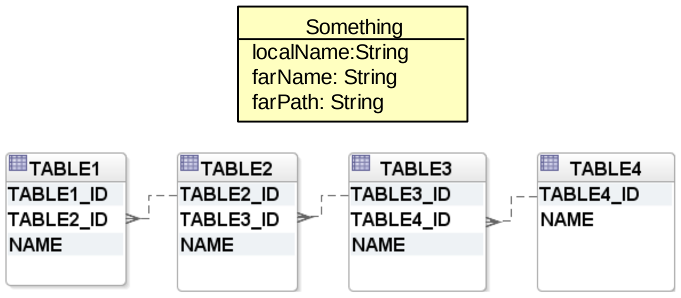


Fig. 4. Far Table linking example- ontology and database schema

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 semantic only to illustrate the design pattern. Tables *table1* to *table2* are linked by two intermediate tables, so four tables are involved. Suppose that OWL ontology consists of one class *Something* that has 3 datatype 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 a table is linked to itself- by two foreign keys based on columns *father_id* and *mother_id*. Figure Fig. [**Fig. 5**] below shows this simple Database shema (one table only) and corresponding OWL ontology presented in MOF style.

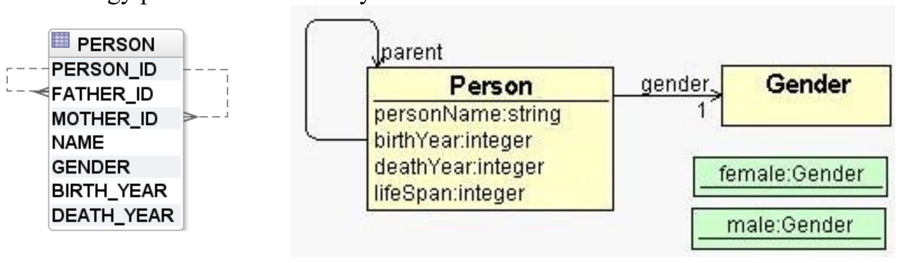


Fig. 5. A RDB schema and ontology of simple Genealogy

Table below lists table data that corresponds to Adam and Eve's posterity taken from [40]

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 fathes or mother is. The ontology has only object property *parent*. The mother or father 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 Relational.OWL platform

Cristian P´erez 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 [6, 7].

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, eg, SPARQL. Relational database data representation in RDF should correspond to relational model. For this purpse relational schema is described in terms of OWL, a central OWL ontology called *Relational.OWL* is created to serve as reference vocabulary. *Relational.OWL* ontology corresponds to relational schema metamodel. It contains classes such as *Database, Table, Column, PrimaryKey* and others. 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 Relatinal.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)
Primārās atslēgas piesaiste tabulai	Funkcionālā propertija „isIdentifiedBy”
Foreign key	Functional property *references* (domain and range being class *Column*)
Maximal data length of column	Property *length*
Precision (decimal digits)	Property *scale*

Most often used classes and propereties of *Relationa.OWL* ontology is illustrated in picture below taken from [45]:

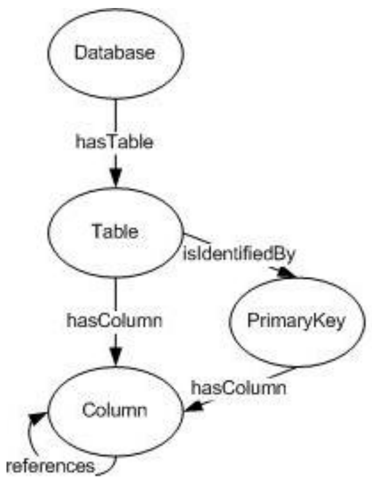


Fig. 6. Relational.OWL Ontology

Full source code for Relational.OWL ontology is given in [44].

It must be noted that *Relationa.OWL* ontology does not allow description of foreign keys based on more than one column because one column references one column. Class *ForeignKey* should be implementated in similar way as of class *PrimaryKey* to allow description of multiple column foreign keys. A relational schema can be expressed in OWL format first describing namespace to Relationa.OWL and then tables belonging to database and columns belonging to tables, see appendix [9.1.2] for details on this and appendix [9.1.3] about RDB data transformation into RDF format.

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. gives base to map relational schema(s) to target ontology by mapping ROWL to target ontology by SPARQL (this is described in section [3.2]).

3.2 Database to target OWL ontology mapping using SPARQL

We 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 paper their paper [6]. The mapping is done as follows. Examples in this section and Fig. 7 are taken from [6].

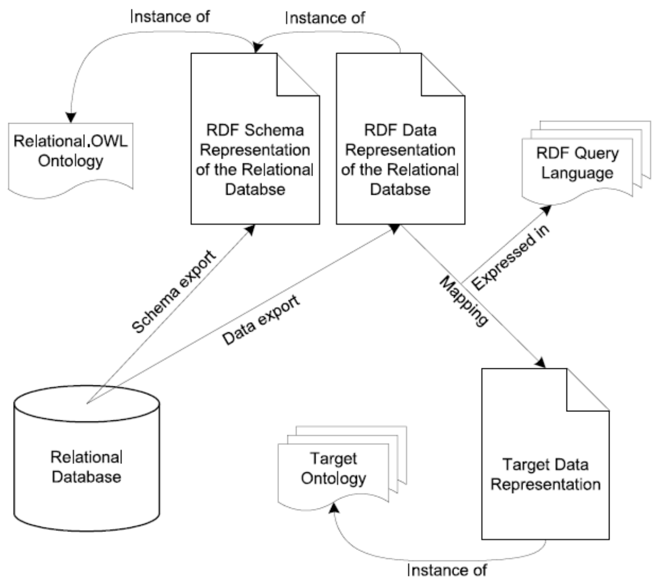


Fig. 7 Mapping process

First Relational Database (RDB) schema is automatically exported into Relational.OWL reprezentation 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 [41] or on the fly by using RDQuery application [42]. 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 [43]. 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 [9.1.6].

RDQuery java application [42] 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 aproach 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 [47] and several tools support them in various degrees [48].

- 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” [49] present a framework where mappings between ontologies are established and SPARQL queries are reformulated.

3.3 D2RQ platform

D2RQ [9], [50] is platform- Treating Non-RDF Relational Databases as Virtual RDF Graphs (Prof. Dr. Chris Bizer form Freie Universität Berlin and colleges). The initial version 0.1 of the framework was issued in June, 2004, the current at the moment 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. The overall architecture is shown in figure below (taken from [9]).

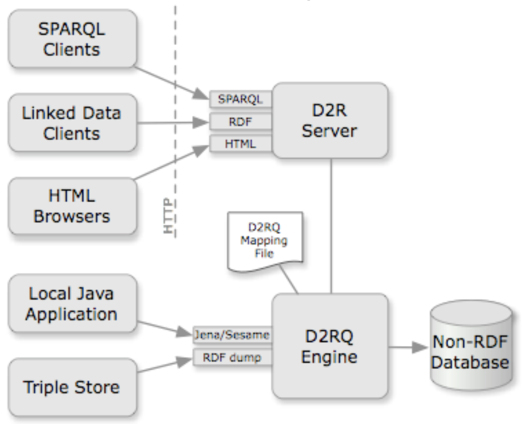


Fig. 8. The architecture of the D2RQ Platform

The mappings are written in D2RQ- a declarative mapping language expressed in RDF and most often written in N3 format. D2RQ Engine is implementated as Jena [51] graph and enables using Jena or Sesame API [52] to get RDF triples by RDF graph processing or executing SPARQL queries. D2RQ mappings are used to translate queries to SQL and translating SQL execution result back as RDF graph. D2R server enables to use web application as SPARQL endpoint or browsing RDF data.

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 getting data on the fly from relational database;
3. java API to use benefits 1. and 2. from java program code;
4. web based SPARQL endpoint.

The main features of D2RQ mapping language are illustrated by code examples below. We start with mini-university example.

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

```
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 2 type of maps: ClassMap and PropertyBridge. The ClassMaps specifies how to get OWL/RDFS class instance triples. Mapping code for *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;
.
```

URI for subject and predicate are specified by pattern in *d2rq:uriPattern.* 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.

PropertyBridge specifies how to get instance triples for datatype or object properties. Property bridge uses `d2rq:belongsToClassMap` to specify classMap for domain. Mapping code for OWL data property *className* specifies that values are to be taken from column *COURSE.NAME*:

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

Object properties uses also *d2rq:refersToClassMap* to specify classMap for range. Mapping for OWL object property *teaches* specifies class maps for domain (belongsToClassMap) and range (refersToClassMap) that are based on tables *TEACHER* and COURSE:

```
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 are used to generate subject part of triples. Range class maps are used to generate object part of triples. `d2rq:join` or `d2rq.condition` can specify table row linking or filtering conditions.

OWL object property *takes* links classes *Student* and *Course* in many to many relation, in DB there correspont 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 class map for *Course* class was defined but not for subclasses *MandatoryCourse* and *OptionalCourse* setting filters on 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*). It is possible to use a better design pattern (taken from D2RQ documentation): specify class maps for sublcasses as Property Bridges for propety *rdf:type* refering to class map for common superclass *map:Course.* Instances of each subclass is determined by row filtering expression. The same design pattern is appropriate for all three subclasses of Teacher class (Assistant, Professor, AssocProfessor).

```
# 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 can have more than one corresponding table (*STUDENT* and *TEACHER* tables for class *PersonID*) so two class maps are needed. Two property bridges are needed for property *IDvalue*:

```
# 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 exampe is given in apendice [9.2].

Now we shown how to use D2RQ mappings to base ontology mapping on far table linking, eg, tables linked by chain of more than 2 foreign key links. The database and target ontology example is described in section [2.3.2].

Several *d2rq.joins* are used to describe table joining. Datatype property value is specified by *d2rq.column* if table field contains required value and *d2rq.sqlExpression* is used if SQL espression is needed to evaluate to get property value (in this example Oracle DB string concatenation operator "||" is used). Full mapping code for the example is given in section [9.2.2]. The code fragment below shows how to fill OWL datatype property *farName* value with column value in table that is reacher 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:

```
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"

Next we show how D2RQ mapping for a case when a table is linked to itself. Database and target ontology example are from genealogy example described in section [2.3.3]. *PERSON* table is linked to itself by two foreign keys based on columns *father_id* and *mother_id*.

D2RQ mapping fragments follows (full code is given in apendice [9.2.3]):

Property bridge for *lifespan* property shows that any SQL expressions can be specified for OWL data property 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 OWL object property *parent* as two foreign *father_id* and *mother_id* keys are corresponding in the database. Those two PropertyBridges connects the same class Person (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 second *person* table reference. Then *d2rq.join* joins *person* to *person*. Such aliasing for joining is appropriate for tables used for range (used in ClassMap referenced from *d2rq.refersToClassMap*). Table used in ClassMap for domain of the property (*d2rq.belongsToClassMap*) should not be changed by aliasing, meaning 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 script can use also translation tables where concrete values coming from database are translates to concrete values (URI or literals) of OWL/RDFS ontology (eg, f→ex:female).

```
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 to specify rdf triple generation in terms of relational database schema but is unaware of target ontology content. Therefore it must be hand coded that could otherwise be infered from the target ontology. For exampe, for object property it is necessary to specify referenced *ClassMaps* for domain and range. If D2RQ mapping language would allow to use the target ontology structure then clauses *d2rq.belongsToClassMap* and *d2rq.rfersToClassMap* could be omitted in typical situations. In exmple below these are odd if OWL object domain and range classes of property enrolled (Student, AcademicProgram) has one ClassMap each.

```
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 folowing scenarios. Suppose we have class *c* with subclasses *C1*, *C2*, …, *Cn* and property *P* that have *c* as domain. The D2RQ mappings it this case can be implemented in two ways.

The first way is to define *PropertyBridges* of property *P* for each sublclass *Ci* referencing 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 much repeated typing: n *ClassMaps* and n *PropertyBridges*.

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 ClassMaps of sublclasses *c1, …cn* as previously.

```
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 ecah 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 that the previous one (no repeated typing) but generates superfluous triples for superclasses. It would be desirable if ClassMap had option to tell if specified triples are to be physically generated. Then we would 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 further that ClassMap for C maps to source database table T, PropertyBridge for property *P1* maps to column *T.C1,* for property *P2* maps to column *T.C2*,… for property *Pn* maps to column *T.Cn*. In case when database schema design is not good one, n can be large (>100). In this situation ClassMap for *C* would normally generate instances for all rows of table *T* when no *d2rq.condition* is set. To specify condition when at least one property instance triple exist for instance of *C*, one would specify lenghty condition which is a tedious work without what superfluous triples would be generated.

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

3.4 R2O Database-to-ontology Mapping language and platform

R2O [Barrasa et al., 2006] [8] is declarative language to express mappings between source RDB schema and target ontology (OWL or XML Schema). The language is XML based in syntax being described in BNF notion. The mappings describe how to obtain ontology class and property instances in terms of source database schema elements. ODEMapster [53] 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 tripless in needed. The engine in implemented as a plugin to NeOn toolkit [54] application, a java desktop Eclipse based application. To enable global access a Web application using ODEMapster should be created.

Both- a source database and target ontology are meant to be independent (pre-existant). Below is picture taken from [8] that illustrate R2O mapping architecture.

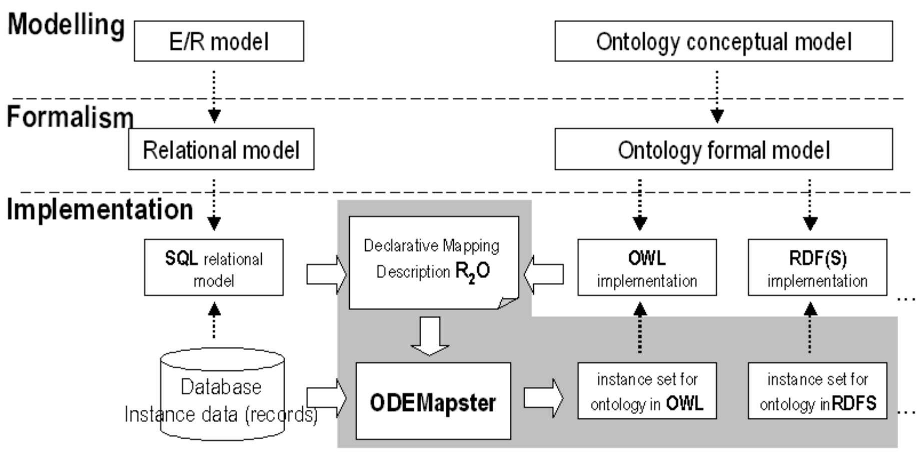

Modelling
E/R model
Ontology conceptual model
Formalism
Relational model
Ontology formal model
Implementation
SQL relational model
Declarative Mapping Description R2O
OWL implementation
RDF(S) implementation
Database Instance data (records)
ODEMapster
instance set for ontology in OWL
instance set for ontology in RDFS

Fig. 9. R2O mapping architecture

R2O language will not be described here, as it is done in [8].The main features will be shortly presented and illustrated by R2O mapping code for mini university example [2.3.1].

R2O mapping contain main structural element description of source RDB (can be more than 1) in <*dbschema-descr*> tag where information about relational DB tables, columns, primary/foreign key are written. Ontology class-to-database mappings are described in <*conceptmap-def*> tag (correspond 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 2 type: transformations (used for URI formation and datatype property values) and condition expressions used, for example, fro <*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 invocation can be used.

OWL/RDFS properties descriptions are included in class mapping desctiption of the domain class- as subtags under <*conceptmap-def*>. Datatype properties are descriped in <*attributemap-def*> tag and Object properties in <*dbrelationmap-def*> where table linking can be described in <*joins-via*> tag which can be omited if mapped tables for classes of domain and range are linked by unique foreign key/primary key link in database.

Below is screenshot of ODEMapster window where part of mapping for simple university example are defined by UI. Only simplest mapping could be defined, no filter (*applies-if*), more than 2 table linkings. These and other custom mappings are to be written manually. ODEMapster supports a subset of the R2O language however it did not open and execute custom mappings written manually.

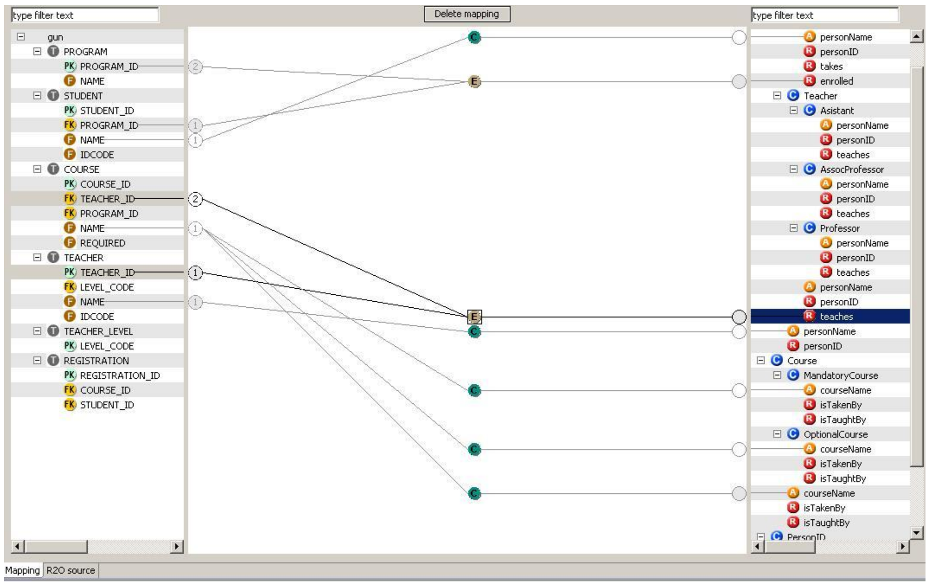


Fig. 10. ODEMapster screenshot for simple university examle mappings

We illustrate now the main points by code fragments for mini-university example. D2O mappings code is longer comparing to D2RQ code for the same example. Full D2O code for the example is given in apendice [9.3].

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

The mapping code is followed by list of <*conceptmap-def*> for each class of ontology. Some has condition description <*applies-if*> (eg, for classes *MandatoryCourse, OptionalCourse, Professor*). For class *Teacher* mapping are as follows

```
<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>
```

First URI pattern ("Teacher" concatenated with value of *TEACHER.TEACHER_ID* field.) As code shows functions (eg, concat) are defined on D2O level and no SQL expresion can be used. Mapping for OWL datatype property (*personName*) is described in *attributemap-def* tag. Next mapping for OWL object property *teaches* is described in *db_relationmap-def* with domain class in *toConcept* attribute and DB table joining described in *joins-via* tag. Mappings for properties are embedded into mappings for classes for domain.

Mapping for OWL Object property *takes* shows multiple table linking example (*STUDENT→REGISTRATION→COURSE*) eg, how to link *STUDENT* table (mapped to domain class) to *COURSE* table (mapped to range class) through intermediate *REGISTRATION* table by means of two column comparison (*eguals*):

```
<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 two tables *STUDENT* and *TEACHER* are 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>
```

A problem is seen there joining 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 can 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. That means no possability to use SQL expressions (RDBMS supported) and use RDB engine to calculate them for row filtering and property value calculation.

D2R mapping language enables to stores RDB shema structure information but no information about the target ontology. Therefore no domain/range information, subclass relation and other ontology information can be used for mapping purposes.

The ODEMapster v.2.2.7 of 02.07.2010 [53] is not mature enough for real life applications- impossible to define row filters, far table links, instability (often crashes with uncatched java Exceptions). The further development of D2O language in unclear. Language specification available is from the original paper [8] from 2004.

3.5 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 automantic Database-to-Ontology Mapping [55]. Here we briefly describe the main points from this paper.

DB2OWL starting point is Relational Database schema structure- Tables, Columns and mainly relations between tables. This structure is analyzed by defined alghoritm to infer appropriate OWL ontology that conform source database design.

The architecture of DB2OWL implementation is shown below in [**Fig. 11**] taken from [56].

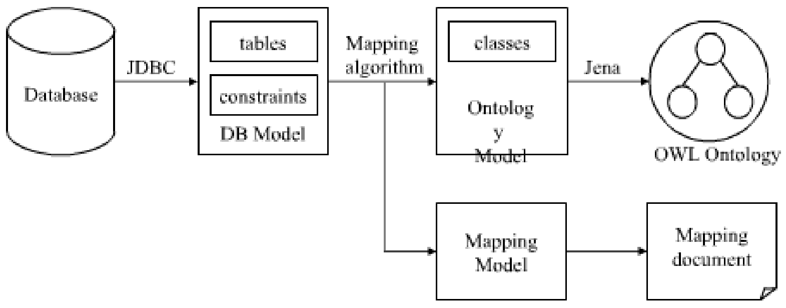


Fig. 11. RDB2OWL framework architecture

Mapping algorithm analyzes database tables and relation types between them and decides about corresponding OWL class and property creation. 3 cases are taken into account:

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 table cases we use database example from mini-university example [2.3.1]

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

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

The mapping algorithm is described by the following steps.

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 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 propery 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 foreign keys are mapped to datatype properties.

Execution of the above mentioned algorithm automatically generates a R2O [8] document that hold the generated mappings between original database and generated ontology. It can be used to translate queries against generated ontology into SQL queries to retrieve corresponding instances.

The described in this section DB2OWL aproach is not appropriate if source database is not well designed (foreign/primary keys not explicitely defined, large tables corresponding to many concept, etc). It should be noted that the correspondence of ontology generated with DB2OWL method is not so strictly conforming to relational model as that generated with Relational.OWL approach [1].

3.6 Vitruoso RDF views

Virtuoso RDF views (C. Blakeley, OpenLink Software, 2007, [10]) is framework that has a mapping specification language between relational database and target OWL/RDFS ontology. Relational database schema/data mapping to OWL/RDF is expressed in “Meta Schema Language”-mapping definition language where quad map patterns are described using SPARQL notion. In typical cases tables are mapped to RDFS classes, table columns to OWL datatype properties and foreign keys to OWL object properties. The mapping language is expressive enough to cope with non-direct custom mapping cases.

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.

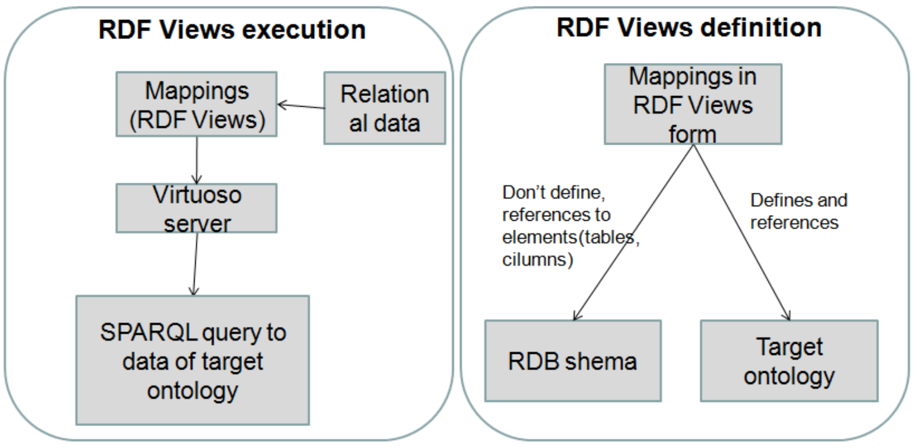


Fig. 12. Virtuoso RDF views architecture

Below some features of the languaga are ilustrated for mini-university example [2.3.1]. The full code are given in appendix [9.3]

The mapping language has means to describe the target ontology. The code belowillustrate description of RDF class, OWL datatype 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 .
…
```

Next IRI classes are defined that are used for URI calculation for triple instances. 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 are both defined in one triple pattern) and reference the defined IRI patterns (course_iri). Code below specifies mapping for class *Course* and OWL data property *courseName* and object property *isTaughtBy*:

```
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)
.
 }
};
```

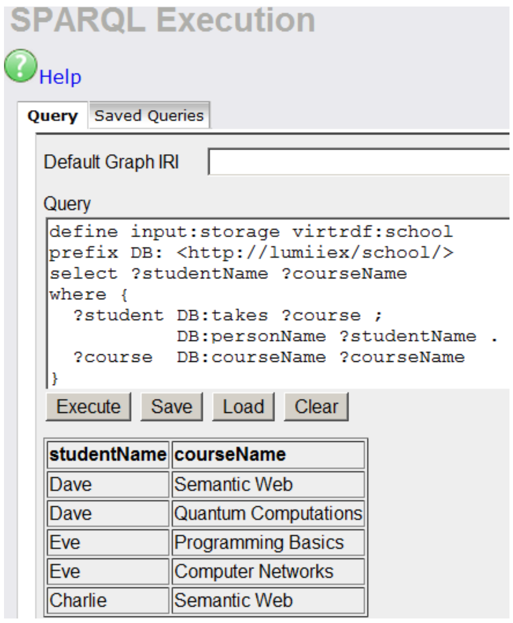
Mappings for subclasses (*MandatoryCourse* as subclass of *Course*) typically reference the same URI patterns as the mappings for superclass (*course_iri*). Additional SQL filter *course.required=1* is added:

```
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 OWL object property *takes* need to specify link chain of 3 tables *student*→*registration*→*course*. It is done in a similar way as one would join tables 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) .
 }
};
```

Finally, 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:



```
define input:storage virtrdf:school
prefix DB: <http://lumiiex/school/>
select ?studentName ?courseName
where {
  ?student DB:takes ?course ;
           DB:personName ?studentName .
  ?course  DB:courseName ?courseName
}
```

studentName	courseName
Dave	Semantic Web
Dave	Quantum Computations
Eve	Programming Basics
Eve	Computer Networks
Charlie	Semantic Web

Fig. 13. SPARQL execution over RDF Views

We make some conclusions about mapping language of Virtuoso RDF views. The mapping language is rather technical one, it takes some effort to learn it (in examples above used only simplest constructions). Custom mapping are more complex, the case when structure database schema and ontology differs significantly. It is not clear how to use SQL expressions for DatatypeProperty value calculation.

Virtuoso opensource version doesn't allow usage of external RDBs with jdbc connections (only built in database can be used).

3.7 Ultrawrap

Ultrawrap (Ultrawrap: Using SQL Views for RDB2RDF by Sequeda, J.F., Cunningham, C., Depena, R., Miranker, D.P. [12]) 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 formulate correct SPARQL queries.

3.8 Triplify

Triplify [13] is a simple approach to publish Linked Data from relational databases that are hidden behind web applications. Triplify offers an adapter and configuration that can be integratted into existing web applications to enable RDF and Linked Data generation by executing defined database SQL views.

The Triplify Web site [57] includes a repository of various Triplify congurations for popular Web applications part of which is third party contributed showing that Triplify is widely used in the industry.

SQL is used as a mapping language where each mapping is expressed as SQL view with special structure (the first columns returns identifiers for instance URIs, columns names are taken for property URI generation, etc). Triples are demanded by means 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) scenarious. For ETL performance improvement some kind of update logs are integrated to enable incremental RDF update.

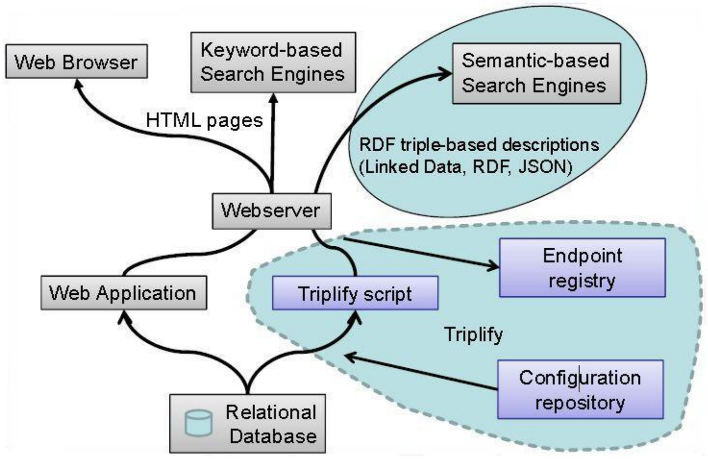


Fig. 14. Triplify overview: the Triplify script is accompanied with a conguration repository and an endpoint registry (picture taken from [13]).

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 *Student* class and subclases of *Course*, *personName* and *courseName* datatype properties and *takes* object property. Mapping burden is laid on column namings (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 that of SQL. SQL as a mapping language is used for performance purpose: to generate more triples faster power of relational databases could be used.

3.9 DartGrid

Dartgrid is a Semantic Web toolkit [33] 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. No special mapping specification language is used. The mappings are defined by help of visual tool. 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. Full-text search capability with concept ranking is provided.

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

There is upcoming W3C standard for direct (technical) mapping [60] 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 accessable. 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 database table data and design: 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 RoRDB w RDF 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, thye are described in document [60]. 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. Also some corner cases of relational schema are taken into consideration eg. 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 DRF nodes.

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

```
@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” .
…
```

One can notice that similar approach of just described direct mapping is also used in Relational.OWL platform [3.1]. But 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.

3.11 R2RML: RDB to RDF Mapping Language (W3C)

There is upcoming W3C standard R2RML [59] for RDB-to-RDF mapping language. The latest version at the moment 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 [2.3.1]. For full mapping source code, see appendix [9.5].

If target ontology class and property is mapped to database table and column, mappings are rather straightforward, eg, 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 mapping can not be expressed simply in terms of simple tables and columns but row filters or calculated values are needed then manual SQL coding is needed. For example, *required* column of COURSE table determine instance of which *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, tripleMaps for all 3 subclasses of *Teacher* class has the same predicateObjectMap. It can be defined in one place and reused many 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 2 tables. Manual SQL coding is to be used for 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 3 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 rather low level language for RDB-to-RDF mappings. Its expressivness relies much on SQL language. Typical cases when data properties are mapped to table columns and object properties- on foreign keys then mappings can be specifies without manual SQL coding. R2RML mapping language is not aware of target ontology therefore ontology structure (eg, domain/range of properties) can not be used to simplify mapping code or enhance expressiveness. R2RML might be thought as a low level SQL oriented technical language. If R2RML eventually has tools for mapping processing ant triple generation then user oriented higher order mapping languages need not to have tools for triple generation. All they need is to have compiler to translate to R2RML.

3.12 Spyder tool

A new emerging mapping language approach is Spyder tool by Revelytix [61]. 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 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 when needed for complex transformations
- simple mapping cases can be expressed simply
- shorter forms of mapping expressions can be obtained by using implicit information (eg., primary/foreign key column list)

The Revelytix RDB Mapping Language shares some ideas with D2RQ but differs from D2RQ in the details. As in D2RQ the Revelytix RDB mapping language specifications are expressed in RDF and written in N3 format. It has also MOF type metamodel which allows 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 expressiveness provided by concrete RDBMS. Example of Revelytix RDB Mapping Language for Mini-University example fragment (classes *Student*, *Course* and properties *courseName* and *takes* (domain: references target ontology entities and db: references source database schema objects)

```
: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 column list:

```
: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). It should be noted that not all database references are expressed by links, for example, criteria strings or join expressions can be written as hand coded SQL text. Although Revelytix RDB mapping language is designed to be user friendly (shorter forms, less repetitions- less errors) it is rather complicated language because it tries to cover models as fully as possible (eg., classes Table, Column, KeyColumn, PrimaryKey, and properties between them). The technically complicated part can be imported while mappings author writes references by hand.

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

Most of existing RDB-to-RDF/OWL mapping approaches such as D2RQ, Virtuoso RDF Views 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.

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

Many RDB-to-RDF/OWL mapping languages are typically 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 (eg. primary key columns; how tables are joined by foreign keys).

Many RDB-to-RDF/OWL mapping languages don't use information about structure of the target ontology (eg, subclass relation) and therefore mapping author needs to repeat that information. For example, information about domain class of some property could allow not to specify the predicate part for that property mapping (eg. not specify *belongsToClassMap* in D2RQ *propertyBridge* specification)

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

4 RDB2OWL mapping specification language

RDB2OWL is a high level declarative RDB-to-RDF/OWL mapping specification language that aimes at solving problems left unsolved or solved partly by other approaches as described in section 3.13.

High level goals of RDB2OWL mapping language are:

- define how elements of target ontology are related to metadata constructs in a source relational database;
- define how instances for target ontology (RDF triples) are gererated from data in source relational database;
- explicit references to elements of source database schema and target ontology are used to make validation, influence and dependency analysis possible;
- provide means for avoiding repetitions (eg, referencing named class maps, usage of implicit foreign key information);
- support to deal with 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 be capable to deal with mapping patterns occurring in real life cases. It is a high level 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). In a typical case mapping expression *expr* specifies that OWL class *c* is mapped to database table and *expr* is written as *DBExpr* (we use this name) annotation to class *c;* mapping expression *expr* specifies the correspondence of OWL data property *p* to database table column and *expr* is written to annotation to property *p;* expression *expr* specifies the correspondence of OWL object property *p* to relation between tables (eg, foreign key) and *expr* is written to annotation to property *p*.

RDB2OWL mapping language has grammar and MOF-style mapping metamodel (that can be re-phrased easily also into a mapping OWL ontology). RDB2OWL implementation is designed as 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 source RDB data.

RDB2OWL mapping language is designed to be compilable to RDB2OWL mapping RDB schema 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 the other mapping languages such as in D2RQ [9], Virtuoso RDF Views [10] or R2RML [16] in order to use tools that support them or will support in the future.

The simple structure of the RDB2OWL mapping metamodel allows 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 [62] (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 [9], [10]. 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 [8], 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 a conceptualization by means of subclasses; or meta-level information tables for grouping table fields into a single multi-valued datatype 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 referred to in object or datatype property mappings.

The RDB2OWL mapping language is devided 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 (eg, foreign key column names). The RDB2OWL Core Plus contain additional advanced constructs (eg, 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 (table row cell values) 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 [36] metamodel in **Fig. 15**, with additional expression and filter metamodel in **Fig. 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 shown in the middle part of **Fig. 15**.

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 itself. 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 datatype 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 datatype and object properties). For the advanced mapping features (in RDB2OWL Core Plus described in section [4.3]) we include, however, the (optional) *subclassOf* information for OWL classes, as well as domain information for OWL properties and range information for OWL object properties.

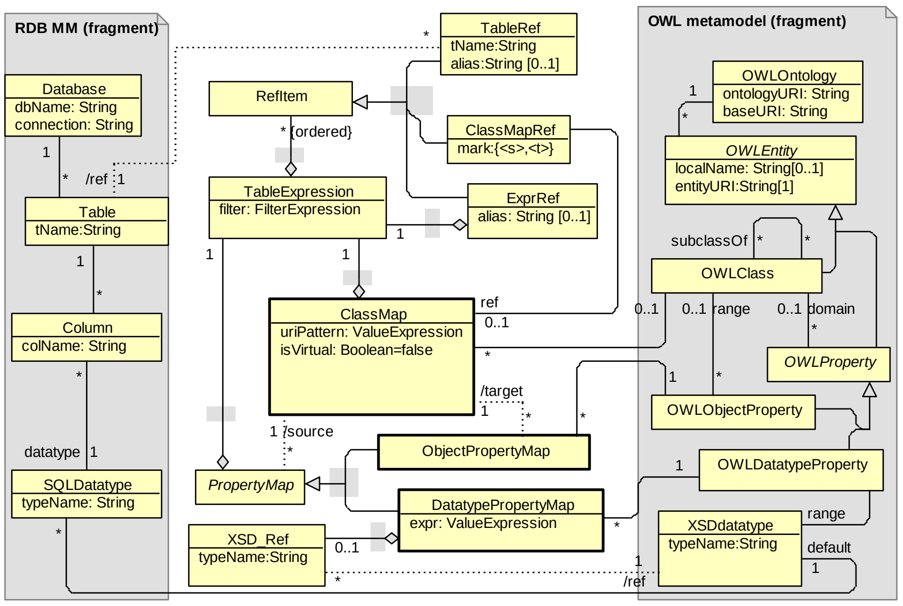


Fig. 15. Raw RDB2OWL Raw mapping metamodel

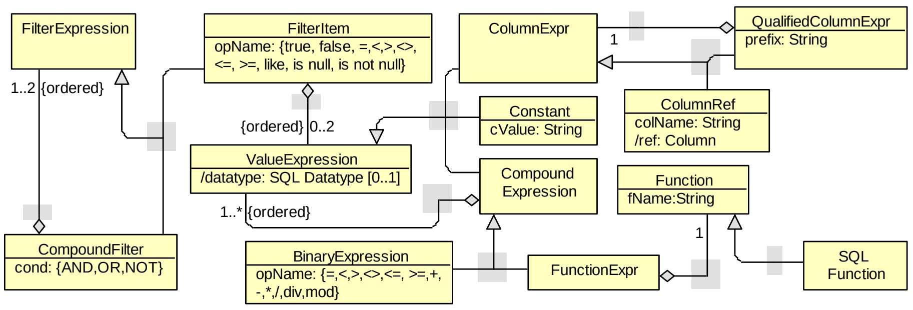


Fig. 16. Expression and filter metamodel

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); datatype 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 datatype property maps may be based on any value expression and not merely on table column. The standard and non standard mapping techniques will be shown on examples further.

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 mapping is done by instances and links of:

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

If an OWL class is mapped to a table context consisting of several tables then class map's TableExpression instance contain more *RefItem* instances of *TableRef* type each referencing one table of the context. The tables are joined by filtering 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.

The *ExprRef* class help building nested table expressions (*TableExpression*→ *ExprRef*→ *TableExpression*). Nesting table expressions are shown in section about RDB2OWL syntax and they are used in transformation steps from RDB2OWL Core to RDB2OWL Raw.

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 datatype property maps. In order to obtain RDF triples for OWL class *C* we require that OWLClass instance for class *C* (localName="C") should be linked to at least one non-virtual class map.

The datatype 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 in **Fig. 16** and specifies value calculation for OWL datatype property. Each datatype property map is based on a source class map (linked to 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*.

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 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*.

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

In typical cases 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 (eg., OWL property has no named class as its domain or range) then source or target class maps can be defined and linked manually by the mapping's author.

For a datatype 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*.

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 datatype or object property may have several annotations describing datatype or object property maps respectively. The syntax structure resembles RDB2OWL Raw metamodel of **Fig. 15**. We present the grammar in simplified form for readability purpose. The detail grammar written in ALTLRWorks tool is given in appendix [9.7]. Its parser java implementation in JavaCC (Java Compiler Compiler JavaCC - The Java Parser Generator) [74] generates instances of syntactic part RDB2OWL full semantic metamodel shown in appendix [9.8]. by using MII REP repository [22] and its java API.

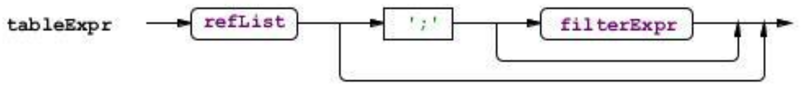
We start mapping syntax explanation by table expressions that is the base for table context definition for class maps and property maps. A table expression description consists of a comma-separated list of reference items, followed by optional filter expression that is separated from the reference item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

The filter expression is built in accordance to the abstract syntax of expression and filter metamodel of **Fig. 16**. Syntax diagrams for EBNF expressions are created in ALTLRWorks tool [73].



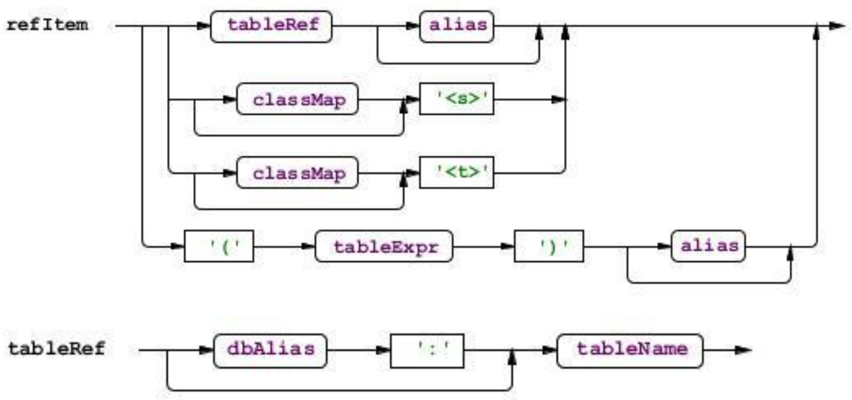


Fig. 17. RDB2OWL Raw table expression syntax

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 source database table column name, 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 expressions are:

```
- STUDENT
- STUDENT S, REGISTRATION R; S.student_id=R.student_id
- STUDENT S, REGISTRATION R, COURSE C;
    S.student_id=R.student_id AND R.course_id=C.course_id
- (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
- <s>, <t>; <s>.teacher_id = <t>.teacher_id
```

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.

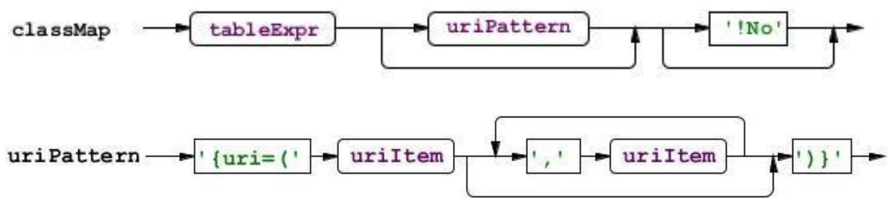


Fig. 18. RDB2OWL Raw class map syntax

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

```
- STUDENT {uri=('Person',student_id)}
- STUDENT {uri=('Person' || student_id)}
- STUDENT S, PROGRAM  P;
    S.program_id=P.program_id
    {uri=('program_' || P.pname, '_student', S.student_id)}
```

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. Some examples of object property descriptions:

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

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 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*.

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

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

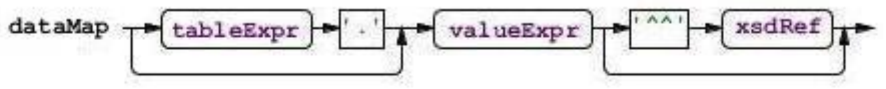


Fig. 19. RDB2OWL Raw data property map syntax

Some examples of datatype properety descriptions are:

- <*s*>.*name*

- *<s>.name^^xsd:String*
- *(Teacher {uri=('Teacher',teacher_id)} <s>).name*
- *(COURSE {uri=('Teacher',teacher_id)} <s>,*
  PROGRAM P; <s>.program_id=P.program_id
).('program: ' || P.name || ', course: ' || <s>.name)
- *((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)

The last of the above expressions is for 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. What was said about inline class map definitions for reference marks <s> and <t> in object property map description can be applied also for datatype property map descriptions but with respect to mark <s> only. In the case when the table expression part for a datatype property map is just *<s>*, we allow omitting it together with the following dot symbol when using 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 is mapped to a table context consisting of several tables then class map's TableExpression instance contain more *RefItem* instances of *TableRef* type each referencing one 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. For 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). In SQL one would write

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

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

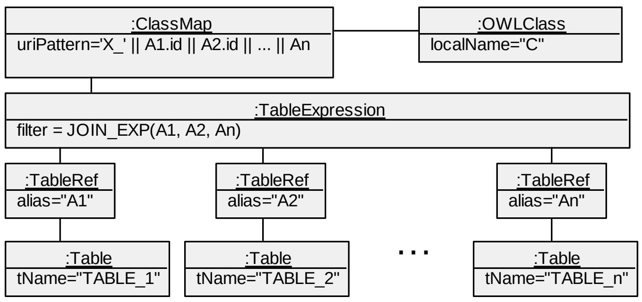


Fig. 20. Class map linked to table context of many joined tables

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

Fig. 21 defines in abstract syntax via RDB2OWL metamodel instances all class maps for the mini-university example (if not stated otherwise examples further will be based on mini-university example). Note that there are two class maps, generating instances of OWL class "PersonID". Note also that the *Teacher* and *Course* classes are based on virtual class maps because instance triples for these classes would be superfluous as their subclasses (*Assistant*, *Professor*, *AssocProfessor* and *MandatoryCourse*, *OptionalCourse*) are based on non-virtual class maps for instance triple generation. The table expressions for subclasses show a typical subclass mapping pattern if their instances correspond to subset of all table rows: superclass and subclass are both mapped to the same table but 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.

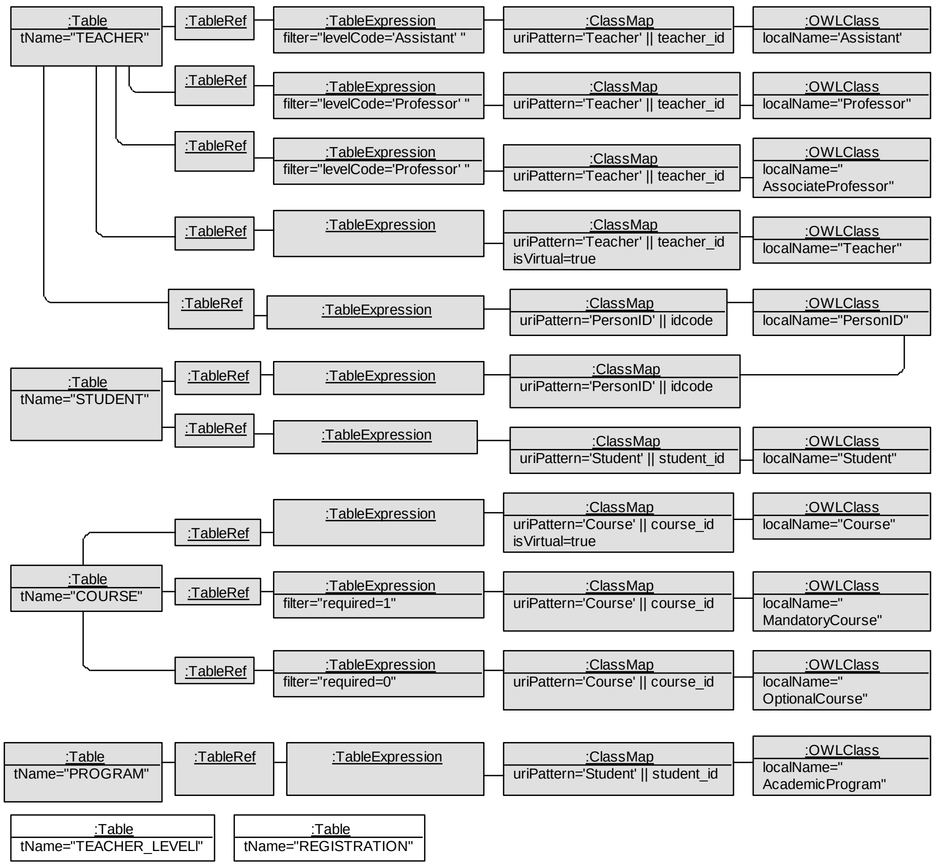


Fig. 21. RDB2OWL Raw Mapping instances for Mini-university example: class maps

Below are three class maps in concrete syntax for *Teacher*, *Course* and *MandatoryCourse* respectively (note 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 not mapped to class maps, in this case normal intension of mapping author, but the model can help to detect omissions by mistake. Other validation features are also possible, eg. un-mapped OWL classes or classes with unintended double class maps.

OWL classes *Teacher* and *Course* may relate to class maps that are based on DB tables *TEACHER* and *COURSE*, respectively. An object property map for the property *teaches* has a table expression with two class map refs (*ClassMapRef*) with aliases <s> and <t> and that has *ref* links to the same class maps that are attached to *Teacher* and *Course* classes respectively, that is, to the source and target class maps. Observe that the property *teaches* is implemented using virtual class maps to the classes *Teacher* and *Course* (the "real" instance generation in *Teacher* and *Course* classes has been specified for their subclasses). Similarly, datatype property map for property *courseName* has a table expression with <s>-marked class map *ref* link to the same class map that is attached to *Course* class (see **Fig. 22** below).

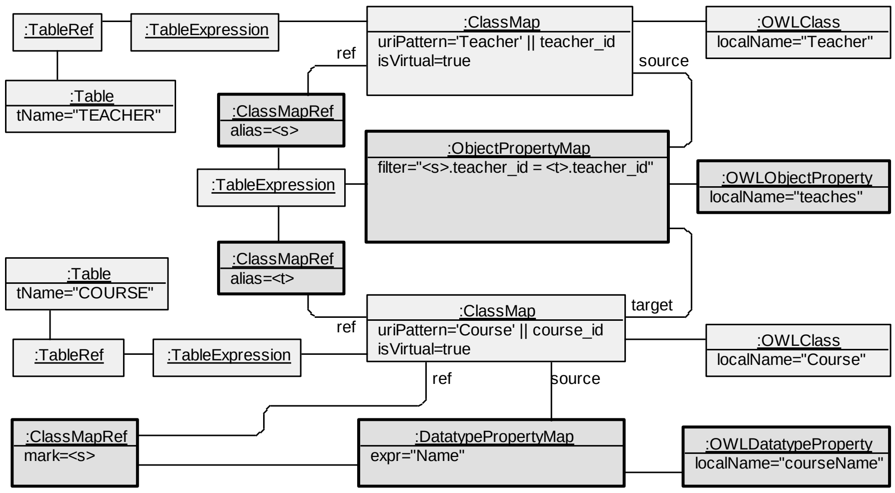


Fig. 22. RDB2OWL Raw Mapping instances for Mini-university example: object and datatype property maps

Concrete syntax for ClassMap, ObjectPropertyMap, ClassMap and DatatypePropertyMap of **Fig. 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 **Fig. 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 to in 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.

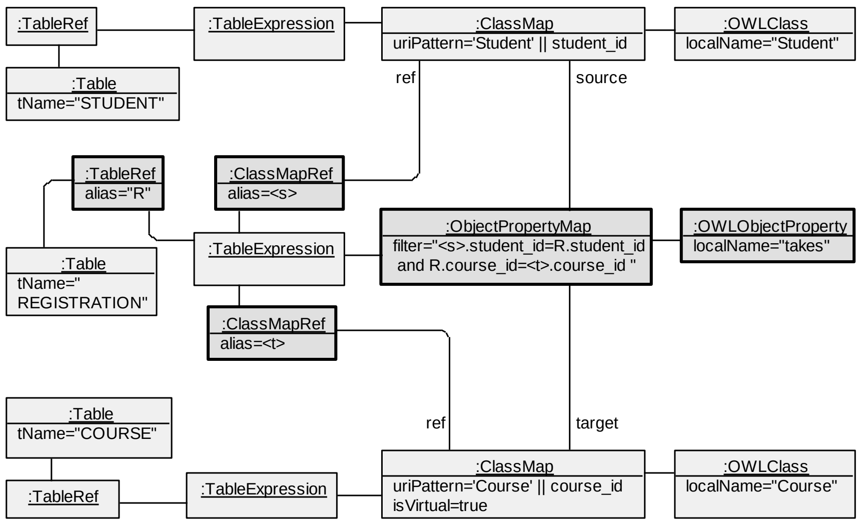


Fig. 23. Mapping instances: property mapping through linked table

There is, however, an alternative solution that is based on not using the "real" class maps for *Student* and *Course* table-to-class mappings, but defining virtual class maps for OWL class *Student* and *Course* URI generation directly from the *REGISTRATION* table (that contains the *student_id* and *course_id* columns that are needed for URI generation). The alternative object property map definition is shown in **Fig. 24**. The down side of this solution is the 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 **Fig. 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.

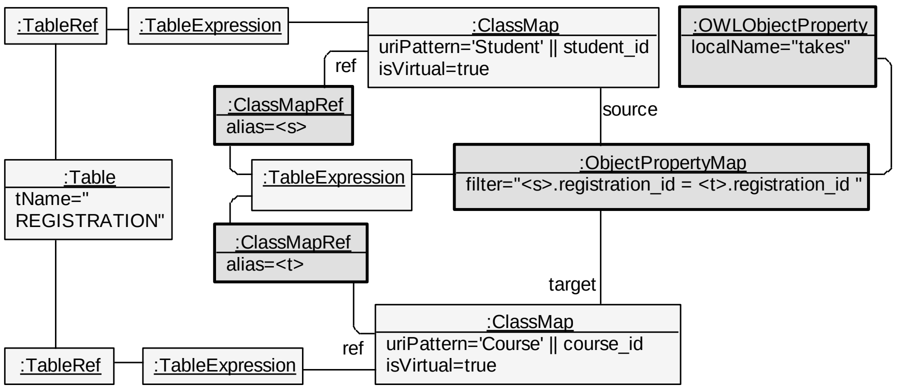


Fig. 24. Linking through table: virtual class maps

Fig. 25 shows how 3 additional tables are joined to datatype property map through it's table expression. The datatype property *farName* from far table linking example [2.3.2] have a table context comprising with 4 tables- one from source class map's table expression and 3 added. Attribute *filter* contain table joining expression and *expr* attribute contain expression- column from the 4-th table TABLE4.

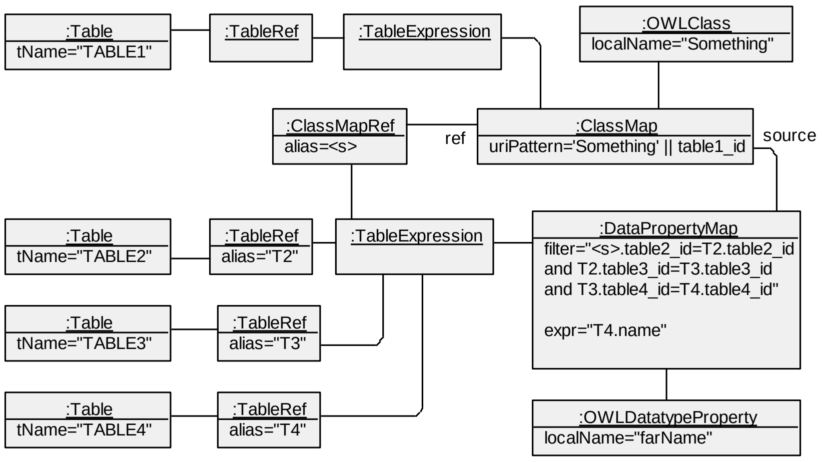


Fig. 25. RDB2OWL Raw Mapping instances for far table linking example

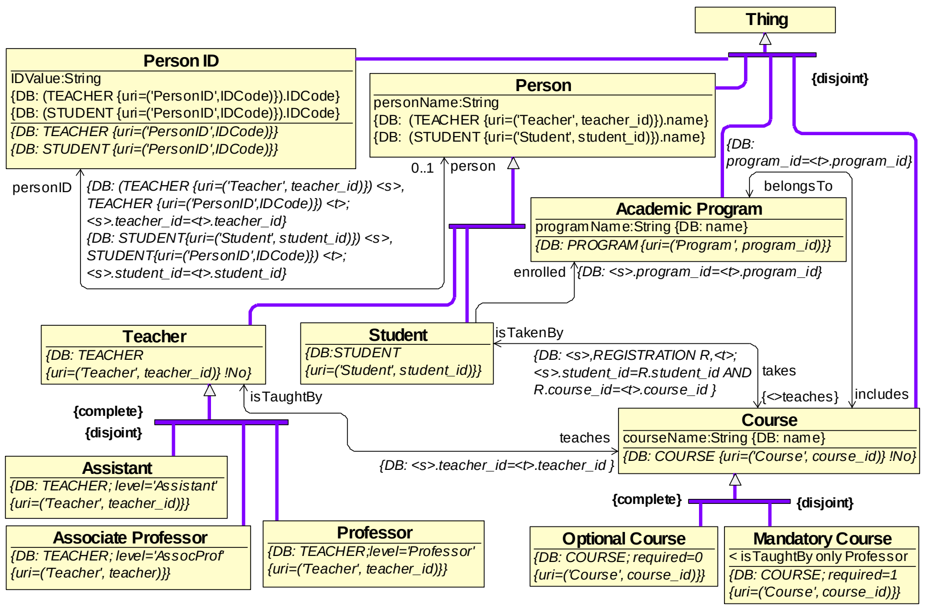
4.1.4 RDB2OWL Raw annotations for Mini-University example

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 **Fig. 26** below. We use here a custom extension of UML-style OWL Graphic Notation editor OWLGrEd [94] 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 datatype 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, a *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 3 subclass class maps (the same for Course class). The foreign key and primary key columns are explicitly written to specify table joining for object property maps, eg., <s>.teacher_id=<t>.teacher_id. It means long typing especially if more than two tables joined, eg. 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.



Thing
{disjoint}
Person ID
IDValue:String
{DB: (TEACHER {uri=('PersonID',IDCode)}).IDCode}
{DB: (STUDENT {uri=('PersonID',IDCode)}).IDCode}
{DB: TEACHER {uri=('PersonID',IDCode)}}
{DB: STUDENT {uri=('PersonID',IDCode)}}
Person
personName:String
{DB: (TEACHER {uri=('Teacher', teacher_id)}).name}
{DB: (STUDENT {uri=('Student', student_id)}).name}
personID
{DB: (TEACHER {uri=('Teacher', teacher_id)}) <s>, TEACHER {uri=('PersonID',IDCode)}) <t>; <s>.teacher_id=<t>.teacher_id}
{DB: STUDENT{uri=('Student', student_id)}) <s>, STUDENT{uri=('PersonID',IDCode)}) <t>; <s>.student_id=<t>.student_id}
0..1
person
{DB: program_id=<t>.program_id}
belongsTo
Academic Program
programName:String {DB: name}
{DB: PROGRAM {uri=('Program', program_id)}}
enrolled
{DB: <s>.program_id=<t>.program_id}
Teacher
{DB: TEACHER {uri=('Teacher', teacher_id)} !No}
Student
{DB:STUDENT {uri=('Student', student_id)}}
isTakenBy
{DB: <s>,REGISTRATION R,<t>; <s>.student_id=R.student_id AND R.course_id=<t>.course_id }
takes
{<>teaches}
includes
isTaughtBy
{complete}
{disjoint}
teaches
Course
courseName:String {DB: name}
{DB: COURSE {uri=('Course', course_id)} !No}
{DB: <s>.teacher_id=<t>.teacher_id }
Assistant
{DB: TEACHER; level='Assistant' {uri=('Teacher', teacher_id)}}
Associate Professor
{DB: TEACHER; level='AssocProf' {uri=('Teacher', teacher)}}
Professor
{DB: TEACHER;level='Professor' {uri=('Teacher', teacher_id)}}
{complete}
{disjoint}
Optional Course
{DB: COURSE; required=0 {uri=('Course', course_id)}}
Mandatory Course
< isTaughtBy only Professor
{DB: COURSE; required=1 {uri=('Course', course_id)}}

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

4.1.5 RDB2OWL Raw Mapping Semantics

By mapping semantic we mean specification how to generate RDF triples from mapping expressions (class maps and property maps) and source database data. It should be taken into consideration for tool developers that process RDB2OWL mapping annotations.

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> …”

Therefore we can assume without loosing 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
$C(t) = \{a.x \mid x \in C(t')\}$;
(c) if *t* consists of items $t_1, \ldots, t_n$ with no aliases specified then
$C(t) = C(t_1) \cup C(t_2) \cup \ldots \cup C(t_n)$; 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 ***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 $t_1, \ldots, t_n$ with no aliases involved then *R(t)* is formed by taking all row combinations from the row sets $R(t_1), \ldots, R(t_n)$.

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* ∊ *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 ***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 ***S***, we form in each case the row set *R(t)* by evaluating *t* on ***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 datatype 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 **Fig. 26**). The table exprsession included is t=*"COURSE; required=1"* and *R(t)* is set of *COURSE* table rows satisfying condition "*required=1*" (see **Table 3** in section [2.3.1]):

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#Course2	http://www.w3.	http://lumii.lv/ex#MandatoryCourse

	org/1999/02/22-rdf-syntax-ns#type	

4.2 RDB2OWL Core

RDB2OWL core language augments RDB2OWL raw metamodel (Fig. 15, Fig. 16) with a possibility to label a reference item (*RefItem* class element) 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 (e.g., only a top row) for columns corresponding to *el* is included into *R(t)* for a combination of values in other *t* columns. Syntactically we write `(Student <s>, Program <t> {top 1 PName asc}; PName >'')` to denote all students and the first program with non-empty name for each of them.

The RDB2OWL core language constructs are summarized in **Fig. 27**. The core language constructs are semantically explained via their translation into the augmented RDB2OWL raw language. The principal novelties here are:

- 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* element) 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.

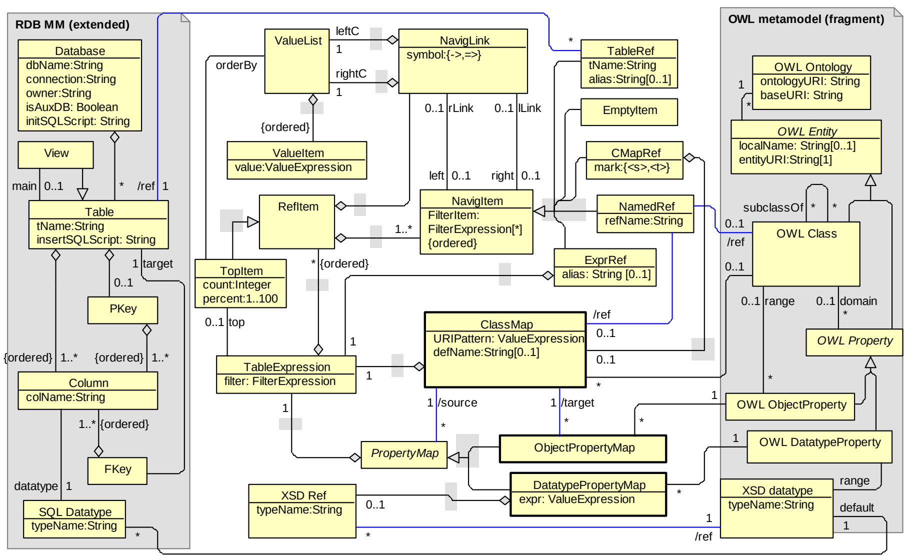


Fig. 27. RDB2OWL Core metamodel

The principal building blocks of the RDB2OWL Core metamodel, as in the model Raw version, 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 encoding filters in simple table expressions as equality conditions 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, as explained below in 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*. The above navigation structure (*), given the mini-University database schema of **Fig.2**, for the property *takes* can then be presented as *<s>=>Registration-><t>*.

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 datatype 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 datatype property map definitions.

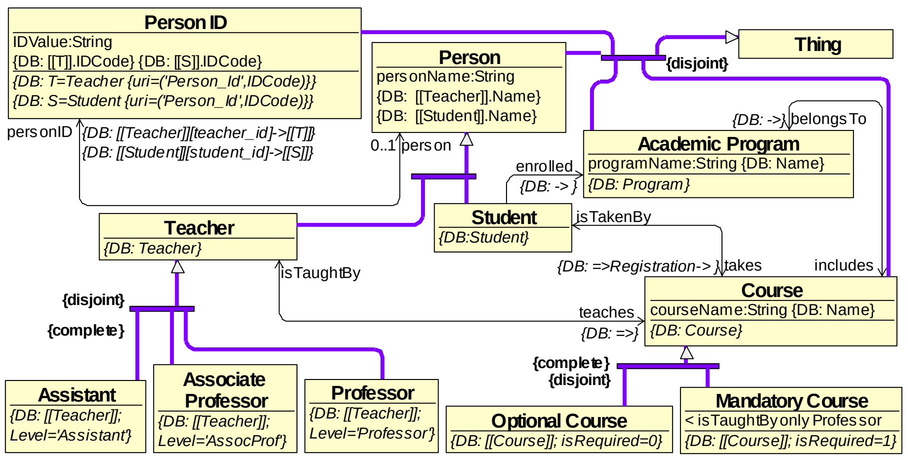

Person ID
IDValue:String
{DB: [[T]].IDCode} {DB: [[S]].IDCode}
{DB: T=Teacher {uri=('Person_Id',IDCode)}}
{DB: S=Student {uri=('Person_Id',IDCode)}}
personID
{DB: [[Teacher]][teacher_id]->[[T]]}
{DB: [[Student]][student_id]->[[S]]}
Person
personName:String
{DB: [[Teacher]].Name}
{DB: [[Student]].Name}
0..1
person
Thing
{disjoint}
{DB: ->}
belongsTo
Academic Program
programName:String {DB: Name}
{DB: Program}
enrolled
{DB: -> }
Teacher
{DB: Teacher}
Student
{DB:Student}
isTakenBy
isTaughtBy
{DB: =>Registration-> }
takes
includes
{disjoint}
{complete}
Course
courseName:String {DB: Name}
{DB: Course}
teaches
{DB: =>}
{complete}
{disjoint}
Assistant
{DB: [[Teacher]]; Level='Assistant'}
Associate Professor
{DB: [[Teacher]]; Level='AssocProf'}
Professor
{DB: [[Teacher]]; Level='Professor'}
Optional Course
{DB: [[Course]]; isRequired=0}
Mandatory Course
< isTaughtBy only Professor
{DB: [[Course]]; isRequired=1}

Fig. 28. Annotated mini-University ontology using RDB2OWL Core model

Figure 7 shows annotated mini-university ontology of **Fig. 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 *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 7.

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*).

Fig. 28 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.

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 [23, 24].

4.3.1 Multiclass Conceptualization

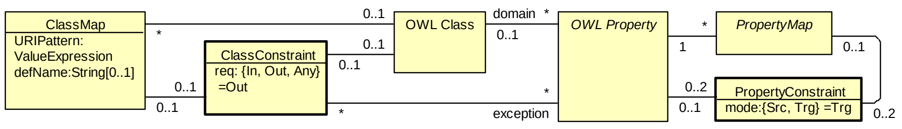


Fig. 29. Class and property constraint metamodel

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 T	OWL Class	OWL Property	Lenghty 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 a *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 datatype properties belonging to these constrained classes.

A *PropertyConstraint* class 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 **Fig. 27**). 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 to be 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 **Fig. 30**, 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.

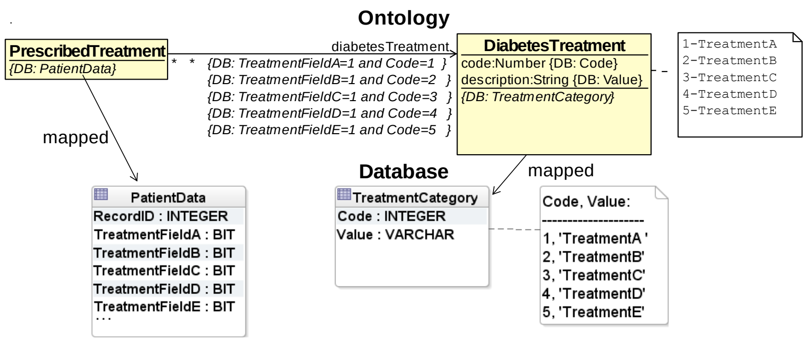


Fig. 30. Ontology and Database fragment for Diabetes Treatment modeling

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 **Fig. 31**.

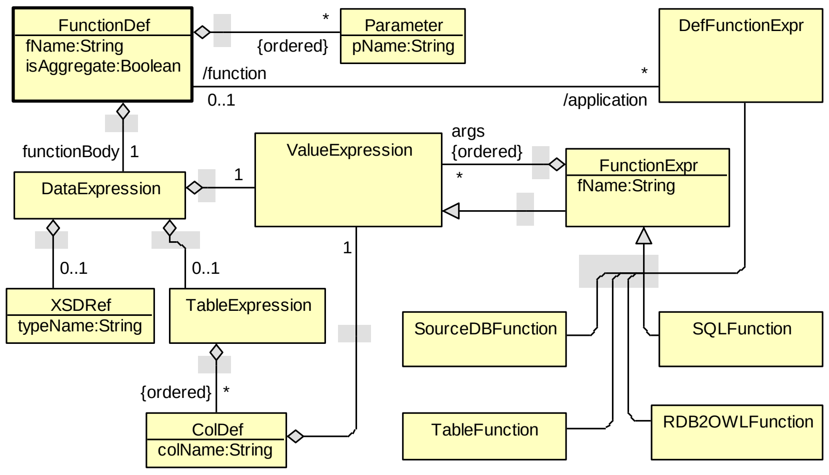


Fig. 31. RDB2OWL Function metamodel

We introduce scalar-argument as well as aggregate functions into RDB2OWL (aggregate functions are shown in **Fig. 32** 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

There are some functions that are frequently needed in different concrete mapping cases. For example, SQL numeric literals 1 / 0 generally are used for boolean *true* / *false* values therefore we have rationale to build-in *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 14. 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 7).

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. Syntactically we have the function definition in the following form:

f(@X$_1$,@X$_2$, ...,@X$_n$)=(T; filter; colDef$_1$, ...,colDef$_m$).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[V$_1$/@X$_1$, ..., V$_n$/@X$_n$], 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[V$_1$/@X$_1$, ..., V$_n$/@X$_n$]* 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: *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 datatype 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 [75]), 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

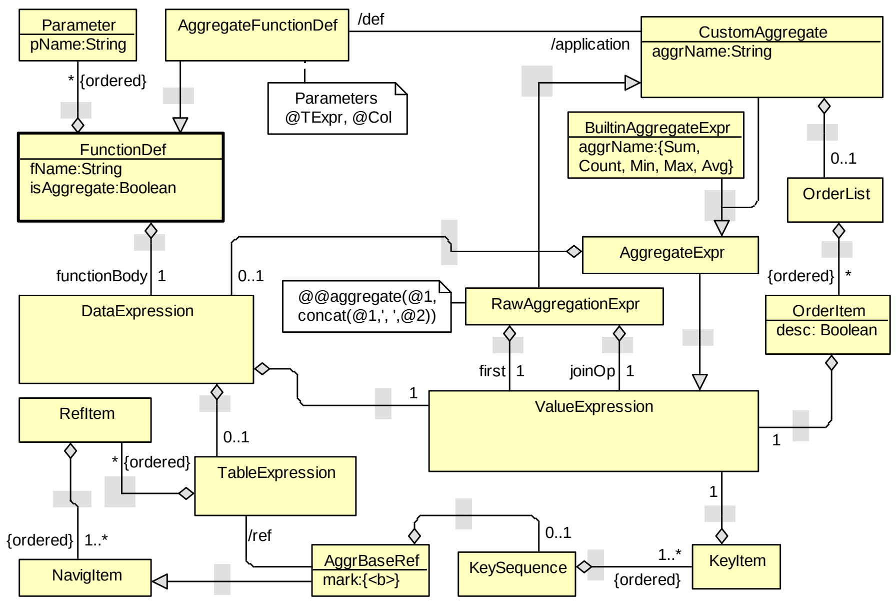


Fig. 32. RDB2OWL Aggregate Function metamodel

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 datatype 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 datatype 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 (recall **Fig 2.** and **Fig. 3.** Mini-university ontology), 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. **Fig. 33** and **Fig. 34** 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 start 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 datatype property map for property *creditsTaken*. Because expression *Sum(=>XRegistration->XCourse.Credits)* omits an explicit base table expression it is assumed to be 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.

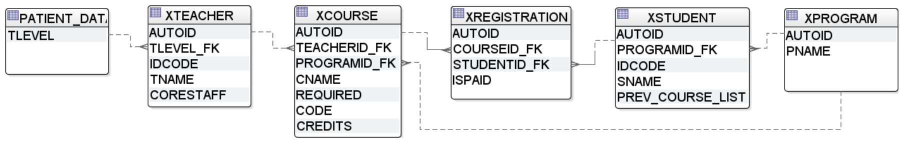
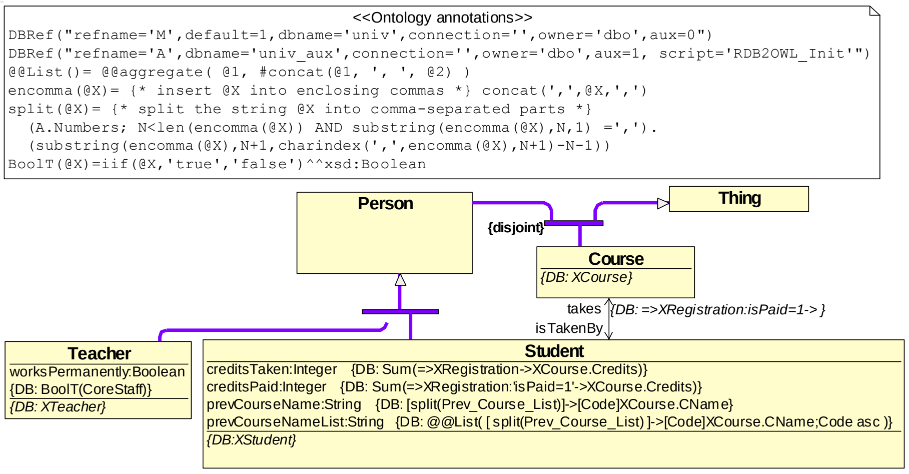


Fig. 33. An extended mini-university database schema



```
<<Ontology annotations>>
DBRef("refname='M',default=1,dbname='univ',connection='',owner='dbo',aux=0")
DBRef("refname='A',dbname='univ_aux',connection='',owner='dbo',aux=1, script='RDB2OWL_Init'")
@@List()= @@aggregate( @1, #concat(@1, ', ', @2) )
encomma(@X)= {* insert @X into enclosing commas *} concat(',',@X,',')
split(@X)= {* split the string @X into comma-separated parts *}
  (A.Numbers; N<len(encomma(@X)) AND substring(encomma(@X),N,1) =',').
  (substring(encomma(@X),N+1,charindex(',',encomma(@X),N+1)-N-1))
BoolT(@X)=iif(@X,'true','false')^^xsd:Boolean
```

Person
Thing
{disjoint}
Course
{DB: XCourse}
takes
{DB: =>XRegistration:isPaid=1-> }
isTakenBy
Teacher
worksPermanently:Boolean
{DB: BoolT(CoreStaff)}
{DB: XTeacher}
Student
creditsTaken:Integer {DB: Sum(=>XRegistration->XCourse.Credits)}
creditsPaid:Integer {DB: Sum(=>XRegistration:'isPaid=1'->XCourse.Credits)}
prevCourseName:String {DB: [split(Prev_Course_List)]->[Code]XCourse.CName}
prevCourseNameList:String {DB: @@List([split(Prev_Course_List)]->[Code]XCourse.CName;Code asc)}
{DB:XStudent}

Fig. 34. 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’}.*

Datatype property map expression for property *prevCourseNameList* is 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

5.1 The mapping execution framework

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

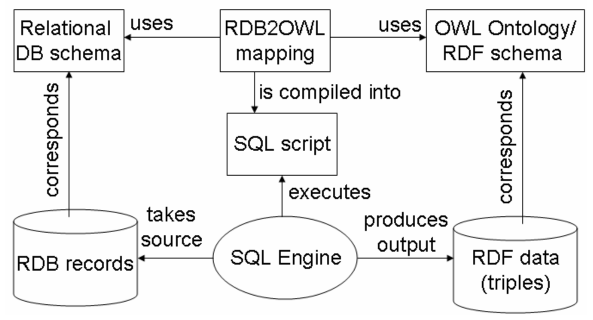


Fig. 35. RDB2OWL framework architecture

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 generatiom 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 by SQL commands to generate another SQL scripts for RDF triples generation. In the second step the generated SQL scripts ar 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

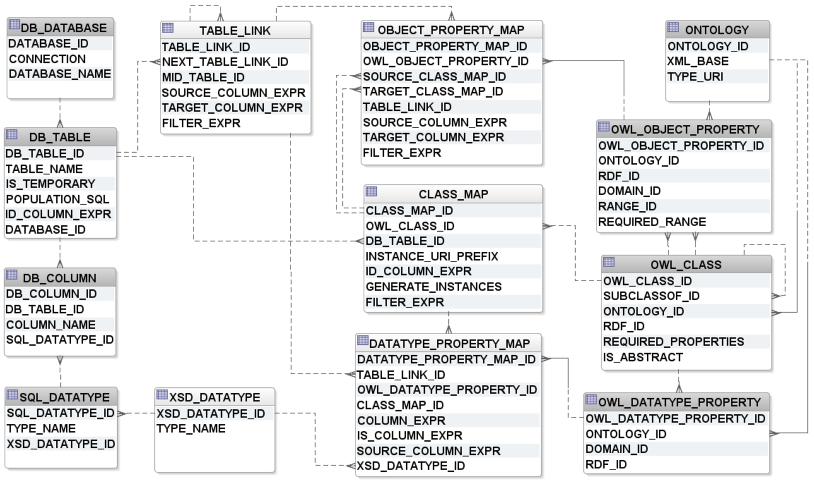


Fig. 36. RDB2OWL mapping RDB schema

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 2 shows relational database schema that stores the RDB2OWL mappings. The mapping information relies on the source database schema description stored in tables *db_database, db_table,* 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 *class_map* record *x* we call *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.

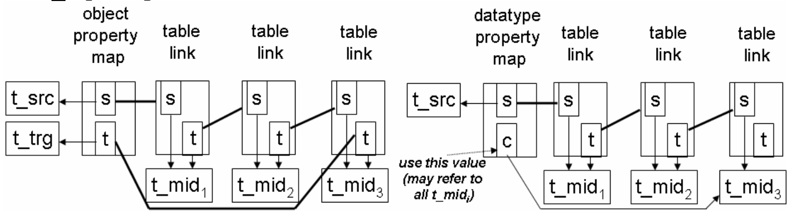


Fig. 37. Intermediate tables in property maps' definitions

The *table_link* table allows introducing intermediate steps in table joining in object property definition, as well as auxiliary linked tables in datatype 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 15. 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 datatype and object property maps (*s* and *t* denote source and target class map id's).

Table 16. 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 datatype 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. 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 datatype 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 datatype property values (we used this pattern for 111 datatype property maps). Auxilary 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

In this section we show triple generation on Mini-university example [2.3.1]. In the next table there are listed OWL class mappings. In the example there are used mappings only to database tables. Below data from tables *class_map* and referenced tables *owl_class, db_table* are listed.

Table 17. 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, however, a few class mappings that are not used in the class instance generation, but which will be further referenced in datatype 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 [9.6] for code) against our sample data, we obtain row set with generated SQL statements, one of which is:

```
SELECT '<http://lumii.lv/ex#Course'
  || course.course_id || '>' as subject,
  '<type>' as predicate,
  '<lumii#MandatoryCourse>' as object
FROM course
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). We use in Table 8 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 18. 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>

Subject	Predicate	Object
<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 19. shows data in table *datatype_property_map* 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 subclasses *Professor, AsocProfessor* and *Asistant* classes. But 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 datatype 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 19. 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 [9.6] for code) against our sample data, we obtain row set with generated SQL statements one of which is:

```
SELECT
  '<lumii#optionalCourse'
  || course.course_id || '>' as subject,
  '<lumii#courseName>' as predicate,
  name as object
```

```
FROM course
WHERE required=0
```

Executing all generated statements in our sample *source DB* we obtain the following triples, duplicates removed (note the abbreviations, as in Table 18):

Table 20. 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 21. 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 7 for more details on referenced *class_map* rows.

Table 21. 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
  <domain_table>_1.<domain_class_map_id→class_map.id_column_expr>,
  <range_table>_2.<range_class_map_id→class_map.id_column_expr>
FROM <domain_table> AS <domain_table>_1
  INNER JOIN <range_table> AS <range_table>_2
    ON <domain_table>_1.<domain_column_expr>
      = <range_table>_2.<range_column_expr>
```

The suffixes _1 and _2 are added here to prevent name collision. 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:

```
SELECT student_1.student_id, student_1.program_id
FROM student AS student_1
INNER JOIN student AS student_2
ON student_1.student_id = student_2.student_id
```

For *enrolled* property the query is

```
SELECT student_1.student_id, program_2.program_id
FROM student AS student_1
INNER JOIN program AS program_2
ON student_1.program_id = program_2.program_id
```

An SQL script for OWL object property instance generation can be defined in a similar way, as for OWL class and OWL datatype property instance generation.

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

```
SELECT
  '<lumii#Program' || program_1.program_id || '>' as subject,
  '<lumii#includes>' as predicate,
  '<lumii#Course' || course_2.course_id || '>' as object
FROM program program_1 INNER JOIN course course_2
  ON program_1.program_id = course_2.program_id
WHERE 1=1 AND 1=1
```

Executing all generated statements in our sample *source DB* we got following triples (note the abbreviations, as in Table 18):

Table 22. 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 tables *student* and *course* through *registration*. 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 *table_link* table is to 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 13 and Table 14 show OWL object property mapping data for properties 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.

After that the corresponding *table_link* data is shown also. *Filter_expr* is not used in the example.

Table 23. 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 24. *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 for code) against our sample data, we obtain row set with generated SQL statements, one of which is:

```
SELECT
'<lumii#Student' || student_1.student_id || '>' as subject,
'<lumii#takes>' as predicate,
'<lumii#Course' || course_2.course_id || '>' as object
 FROM student student_1
INNER JOIN registration registration_3
  ON student_1.student_id = registration_3.student_id
INNER JOIN course course_2
  ON registration_3.course_id = course_2.course_id
WHERE 1=1 AND 1=1 AND 1=1 AND 1=1
```

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

Table 25. 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>
<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 our example database we get following triple set essentially being data export from original relational database to RDF format for target OWL ontology. It is union of data showed in Tables 18, 20, 22 and 25 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>
```

5.5 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.

For example list of OWL classes without class maps can be returned by simple SQL:

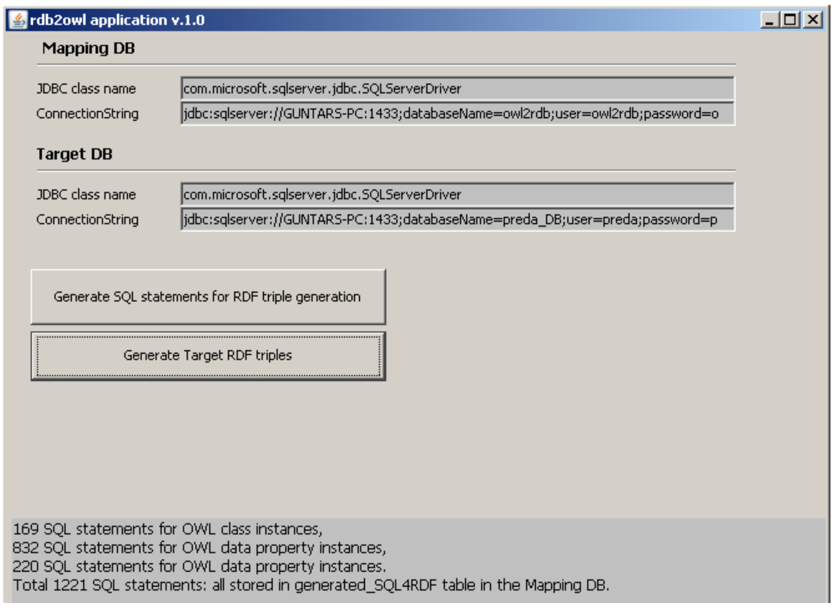
```
SELECT rdf_id AS class FROM owl_class c
WHERE NOT EXISTS
(
  SELECT 1 FROM class_map cm WHERE c.owl_class_id=cm.owl_class_id
) AND c.is_abstract=0
```

A list of OWL datatype properties without maps can be abtained by folowing SQL:

```
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
 )
```

5.6 Implementation as java application

The RDF triple generation two step processes (**Fig. 35**) we implementated as user interface java application. When connection information to the mapping and source databases are entered then both processes can be executed by pressing corresponding command button.


rdb2owl application v.1.0

Mapping DB

JDBC class name	com.microsoft.sqlserver.jdbc.SQLServerDriver
ConnectionString	jdbc:sqlserver://GUNTARS-PC:1433;databaseName=owl2rdb;user=owl2rdb;password=o

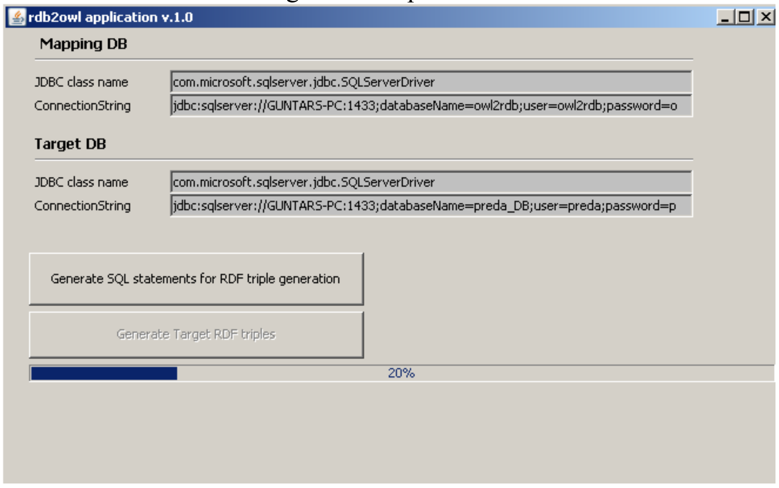
Target DB

JDBC class name	com.microsoft.sqlserver.jdbc.SQLServerDriver
ConnectionString	jdbc:sqlserver://GUNTARS-PC:1433;databaseName=preda_DB;user=preda;password=p

Generate SQL statements for RDF triple generation
Generate Target RDF triples

```
169 SQL statements for OWL class instances,
832 SQL statements for OWL data property instances,
220 SQL statements for OWL data property instances.
Total 1221 SQL statements: all stored in generated_SQL4RDF table in the Mapping DB.
```

Press button <Generate target RDF triples> and wait...


rdb2owl application v.1.0

Mapping DB

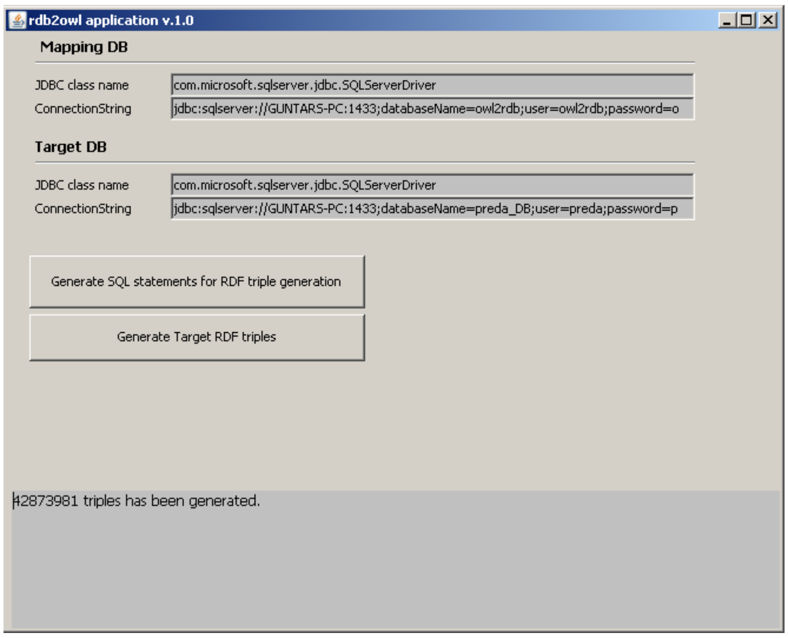
JDBC class name	com.microsoft.sqlserver.jdbc.SQLServerDriver
ConnectionString	jdbc:sqlserver://GUNTARS-PC:1433;databaseName=owl2rdb;user=owl2rdb;password=o

Target DB

JDBC class name	com.microsoft.sqlserver.jdbc.SQLServerDriver
ConnectionString	jdbc:sqlserver://GUNTARS-PC:1433;databaseName=preda_DB;user=preda;password=p

Generate SQL statements for RDF triple generation
Generate Target RDF triples
20%

when the process done:


rdb2owl application v.1.0
Generate SQL statements for RDF triple generation
Generate Target RDF triples

Mapping DB

JDBC class name: com.microsoft.sqlserver.jdbc.SQLServerDriver
ConnectionString: jdbc:sqlserver://GUNTARS-PC:1433;databaseName=owl2rdb;user=owl2rdb;password=o

Target DB

JDBC class name: com.microsoft.sqlserver.jdbc.SQLServerDriver
ConnectionString: jdbc:sqlserver://GUNTARS-PC:1433;databaseName=preda_DB;user=preda;password=p

```
42873981 triples has been generated.
```

5.7 Latvian Medicine Registries: A Case Study

The data mapped from RDB to OWL format consists of 6 Latvian medical registries (Sugar registry, Multiple screlosis registry, Injury registry, Mental inlness registry, Cancer regsitry and Narcotic registry) [23],[24], including 106 source database tables, 1353 table columns and total more that 3 million rows, altogether 3 GB of data. The corresponding OWL ontology had 172 OWL classes, 814 OWL datatype properties and 218 OWL object properties.

The mapping has been implemented on a laptop computer with Intel Mobile Core 2 Duo T6570 processor running Windows Vista, 3 GB of RAM. The mapping DB as well as source DB (Medicine DB) was served by Microsoft SQL Server 2005. 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, 8 minutes for indexing, 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).

Timing details and statistics follows.

Table 26. Element counts in target ontology

Item	Count
OWL Classes (non abstract)	168
OWL datatype properties (domain non abstract)	810
OWL object properties (domain and rangr non abstract)	198

Total objects	1176

Table 27. 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 28. Generated SQL counts for RDF generation

Item	Count
SQL for OWL class instances	169
SQL for OWL datatype property instances	832
SQL for OWL object property instances	220
Total objects	1221

Table 29. Source DB (database file size: 3G)

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

Table 30. Generated triples

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

Table 31. Timings for triple generation

Step	**Time (min:sec)**
RDF triple generation (42,84 milj)	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.

6 Overall implementation architecture RDB2OWL language

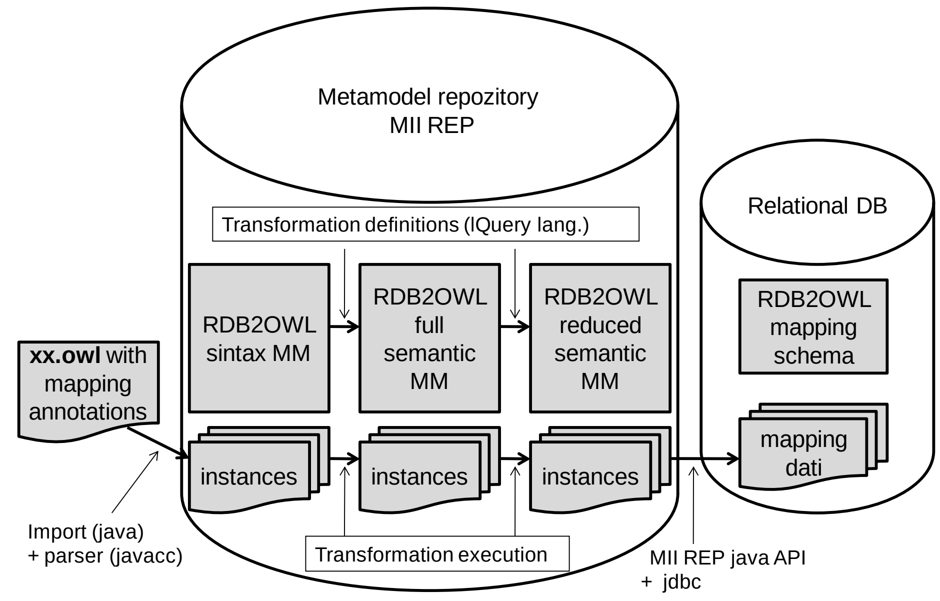


Fig. 38. Ite implementation architecture of RDB2OWL language

Fig. 38 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 are generated as described in sections 5.1 and 5.2.

The process starts by reading the target ontology annotated with RDB2OWL mapping language. The annotations are parsed (javacc [74]) into RDB2OWL metamodel instances in repository. These instances correspond to mere syntactic structure of mapping expressions. Then these instances are analysed to add 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 process is starter 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 explicite columns name omissions in navigation links. At the end of semantic analysis RBB2OWL semantic MM instances are obtained. At this point metamodel the instances correspond to RDB2OWL Core Plus language. Then another transformation step converts the metamodel instances from RDB2OWL Core Plus level in to RDB2OWL Raw language level: changing navigation links into reference item lists, changes high level constructs into lower, for example, changes function calls into basic expressions (merging callers table expression with table expression of the called function). The last step is to transform the RDB2OWL Raw metamodel instances into RDB2OWL mapping schema data, from which RDB2OWL triples can be generated.

The full RDB2OWL language metamodel is packed in appendix [9.8].

The associations and classes displayed in solid lines comprise a full syntax metamodel whose instances can be obtained by parsing of RDB2OWL annotations (using javacc). The classes and associations denoted by dotted lines comprise full semantic metamodel (RDB2OWL core language). Its instances can be calculated by analyzing instances of full syntax metamodel.

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.

After full syntax metamodel instances are generated the transformation steps are executed that reduces the metamodel to *RDB2OWL raw* level. It is done mainly by transformation navigation item links to table ref item lists (e.g. "Course->Teacher" to "Course C, Teacher T; C.TeacherID_FK=T.AutoID").

7 Conclusion

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 means, 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 thesis work was about make relational databases accessable to semantic web technologies, and particulary, 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 and with high level constructs.
- Created RDB2OWL language syntax parser.
- RDB2OWL mapping implementation was developed 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-engeneering of Latvia Medicine 6 registry databases where 42 milj. triples were generated in 18,5 minutes (without dump export to text file).
- Ontology of Latvia Medicine registries was annotated with RDB2OWL mappings language expressions showing that the language is aplicable to pactical industry use_case.

The development of Semantic analyser is in progress that transformes the domain ontology annotations into REB2OWL mapping schema for execution. One of future perspectives is to create compiler from RDB2OWL into emerging 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.

8 References

1. Resource Description Framework (RDF), http://www.w3.org/RDF/
2. RDF Vocabulary Description Language: RDF Schema, http://www.w3.org/TR/rdf-schema/
3. Web Ontology Language (OWL), http://www.w3.org/2004/OWL/
4. OWL 2 Web Ontology Language, Structural Specification and Functional-Style Syntax http://www.w3.org/TR/owl2-syntax/
5. W3C RDF Validation Service http://www.w3.org/RDF/Validator/
6. 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
7. Christian Perez de Laborda and Stefan Conrad: Database to Semantic Web Mapping using RDF Query Languages LNCS 4215, pp. 241-254. Springer, Heidelberg, 2006.
8. 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.
9. D2RQ Platform. Treating Non-RDF Relational Databases as Virtual RDF Graphs. http://www4.wiwiss.fu-berlin.de/bizer/D2RQ/spec/
10. C.Blakeley: “RDF Views of SQL Data (Declarative SQL Schema to RDF Mapping)”, OpenLink Software, 2007.
11. 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.
12. 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)
13. 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).
14. W3C RDB2RDF Working Group, http://www.w3.org/2001/sw/rdb2rdf/
15. A Survey of Current Approaches for Mapping of Relational Databases to RDF, http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf
16. R2RML: RDB to RDF Mapping Language, http://www.w3.org/TR/r2rml/
17. A Direct Mapping of Relational Data to RDF, http://www.w3.org/TR/2011/WD-rdb-direct-mapping-20110324/
18. 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.
19. 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.
20. 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
21. W3C SWEO Linking Open Data community project URL: http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

22. 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.
23. 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).
24. 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.
25. 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.
26. OWLGrEd, http://owlgred.lumii.lv/
27. Ontology Definition Metamodel. OMG Adopted Specification. Document Number: ptc/2007-09-09, November 2007. http://www.omg.org/docs/ptc/07-09-09.pdf
28. 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.
29. Open Government Directive of December 8, 2009: http://www.whitehouse.gov/sites/default/files/omb/assets/memoranda_2010/m10-06.pdf
30. The UK public data website, http://data.gov.uk
31. TED2009 conference, URL: http://conferences.ted.com/TED2009/
32. T.Berners-Lee: Relational Databases on the Semantic Web. http://www.w3.org/DesignIssues/RDB-RDF.html, 1998.
33. 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
34. Spyder tool, URL: http://www.revelytix.com/content/spyder
35. Semantic SQL: http://semanticsql.com/
36. OMG's MetaObject Facility, http://www.omg.org/mof/
37. MOF QVT, http://www.omg.org/spec/QVT/1.0/
38. MOLA resources. URL:http://mola.mii.lu.lv/
39. The <AGG> Homepage, http://user.cs.tu-berlin.de/~gragra/agg/
40. Data for Adam and Eve's Posterity. http://www.johnpratt.com/items/docs/adam_gen/adam.html#
41. Relational.OWL application http://sourceforge.net/projects/relational-owl/
42. RDQuery application http://sourceforge.net/projects/rdquery/
43. Pellet, http://clarkparsia.com/pellet
44. Relational.OWL ontology http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#
45. Relational.OWL Application documentation http://dbs.cs.uni-duesseldorf.de/RDF/docs/ROWLApp/
46. Renato Iannella. Representing vCard Objects in RDF/XML. http://www.w3.org/TR/vcard-rdf, 2010. W3C Member Submission 20 January 2010.
47. SPARQL 1.1 Query Language, http://www.w3.org/TR/sparql11-query/
48. SPARQL New Features and Rationale. W3C Working Draft 2 July 2009 http://www.w3.org/TR/sparql-features/
49. 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

50. 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/
51. Jena- A Semantic Web Framework for Java. http://jena.sourceforge.net/
52. Sesame- Java based framework for storage, inferencing and querying of RDF data. http://www.openrdf.org/
53. ODEMapster engine. url: http://neon-toolkit.org/wiki/ODEMapster
54. NeOn toolkit. url: http://neon-toolkit.org
55. 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
56. R. Ghawi, N. Cullot: Database-to-Ontology Mapping Generation for Semantic Interoperability. A slideshow, http://www.slideshare.net/rajighawi/db2owl
57. Triplify website, URL: http://Triplify.org.
58. W3C RDB2RDF Incubator Group, http://www.w3.org/2005/Incubator/rdb2rdf/
59. R2RML: RDB to RDF Mapping Language, http://www.w3.org/TR/r2rml/
60. A Direct Mapping of Relational Data to RDF, http://www.w3.org/TR/2011/WD-rdb-direct-mapping-20110324/
61. Spyder tool, URL: http://www.revelytix.com/content/spyder
62. 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.
63. G.Bumans, Mapping between Relational Databases and OWL Ontologies: an Example, to appear in Scientific Papers of University of Latvia, Computer Science and Information Technologies, 2010.
64. 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.
65. E.Kalnina, A.Kalnins, E.Celms, A.Sostaks: Graphical template language for transformation synthesis. In: M. van den Brand, D.Gašević, J.Gray (Eds.), Proceedings of Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009 Revised Selected Papers, LNCS 5969, pp. 244--253. Springer, Heidelberg, 2010.
66. 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.
67. OpenLink Virtuoso Platform. Automated Generation of RDF Views over Relational Data Sources. URL: http://docs.openlinksw.com/virtuoso/rdfrdfviewgnr.html
68. Object Management Group MOF QVT Final Adopted Specification. URL: http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf
69. ATLAS Model Transformation Language. URL:http://www.eclipse.org/m2m/atl/
70. Eclipse Modeling Framework Project (EMF). URL: http://www.eclipse.org/modeling/emf/
71. Semantic SQL: http://semanticsql.com/
72. Linked data: http://linkeddata.org/
73. ANTLRWorks: The ANTLR GUI Development Environment, URL: http://www.antlr.org/works/index.html

74. Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator, URL: http://javacc.java.net/
75. J.Moden. The "Numbers" or "Tally" Table: What it is and how it replaces a loop. http://www.sqlservercentral.com/articles/T-SQL/62867/

9 Apendice

9.1 Relational.OWL platform

9.1.1 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 [6],[7]. 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>
```

9.1.2 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 [9.1.1].

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 [9.1.1]

9.1.3 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 [9.1.1]). 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 [9.1.5]

9.1.4 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>
```

9.1.5 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>
```

9.1.6 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 32. 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  .
  }
}
```

9.2 D2RQ platform

9.2.1 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;
    .
```

9.2.2 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";
 .
```

9.2.3 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;
.
```

9.3 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  .
 }
};
```

9.4 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>
```

9.5 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" ]
    ];
    .
```

9.6 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. They also use MsSQL functions ISNULL, REPLACE and others. It is easy to rewrite these SQLs for another DB if needed.

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
```

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
```

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
```

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
   'SELECT '
   || '''<' || o.xml_base
   ||  cm_domain.instance_uri_prefix || ''''
   || ' || '  || c2t_domain.table_name || '_1'
   || '.' || cm_domain.id_column_expr || ' || ''>'' as subject'

   || ',''' || '<'|| o.xml_base || op.rdf_id  || '>'' as predicate'

   || ',' || '''<' || o.xml_base
   ||  cm_range.instance_uri_prefix || ''''
   || ' || '  || c2t_range.table_name || '_2'
   || '.' || cm_range.id_column_expr || ' || ''>'' as object'
```

```
  || ' FROM '
  || c2t_domain.table_name || ' ' || c2t_domain.table_name || '_1'
  || ' INNER JOIN '
  || tl.mid_table_name || ' ' || tl.mid_table_name || '_3'
  || ' ON ' || c2t_domain.table_name || '_1. '
  || opm.source_column_expr
  || ' = ' || tl.mid_table_name || '_3.' || tl.source_column_expr
  || ' INNER JOIN '
  || c2t_range.table_name || ' ' || c2t_range.table_name || '_2'
  || ' ON ' || tl.mid_table_name || '_3.' || tl.target_column_expr
  || ' = ' || c2t_range.table_name || '_2.' || opm.target_column_expr
  || ' WHERE ' || NVL(cm_domain.filter_expr ,' 1=1 ')
  || 'AND    ' || NVL(cm_range.filter_expr , ' 1=1 ')
  || 'AND    ' || NVL(tl.filter_expr , ' 1=1 ')
  || 'AND    ' || NVL(tl.filter_expr , ' 1=1 ')
  AS generated_SQL
FROM
  owl_object_property op,  ontology o,
  object_property_map opm, class_map cm_domain,
  class_map cm_range,            class2table c2t_domain,
  class2table c2t_range,         table_link tl
WHERE
 op.ontology_id=o.ontology_id AND
  op.owl_object_property_id=opm.owl_object_property_id AND
  opm.domain_class_map_id =cm_domain.class_map_id AND
  opm.range_class_map_id =cm_range.class_map_id AND
  cm_domain.class2table_id=c2t_domain.class2table_id AND
  cm_range.class2table_id=c2t_range.class2table_id AND
  opm.table_link_id=tl.table_link_id AND
  opm.table_link_id IS NOT NULL AND op.ontology_id=1
```

9.7 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:',';
```

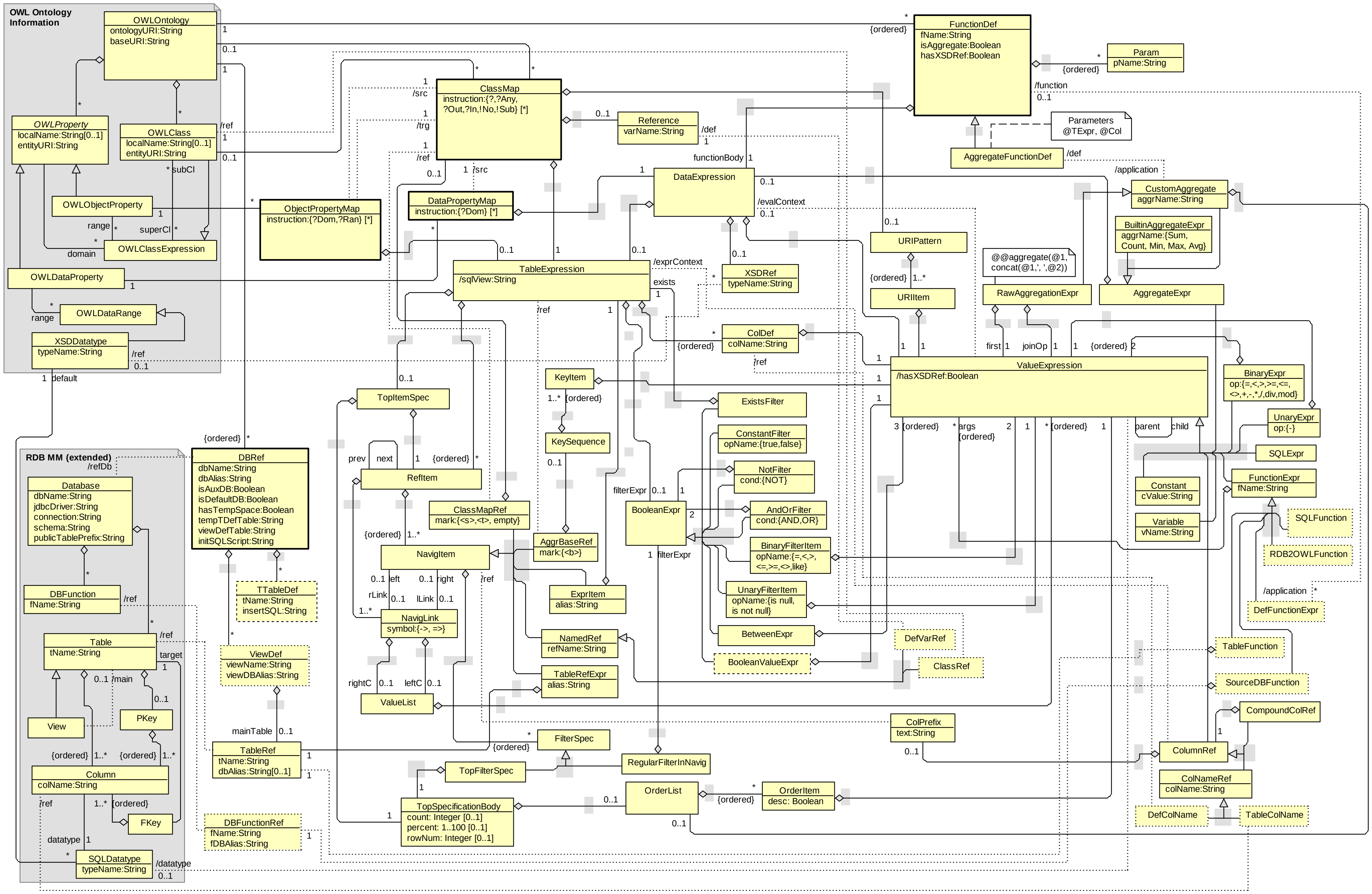
```
ZEROONE :'0'|'1';
INT : '0'..'9'+;
COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    |   '/*' ( options {greedy=false;} : . )* '*/'
{$channel=HIDDEN;};
WS  :   ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;} ;
STRING :  '\'' ( ESC_SEQ | ~('\\'|'\'') )* '\'';
XSD_TYPE_PREFIX: 'xsd:';
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;
```

9.8 RDB2OWL full semantic metamodel



OWL Ontology Information
OWLOntology
ontologyURI:String
baseURI:String
OWLProperty
localName:String[0..1]
entityURI:String
OWLClass
localName:String[0..1]
entityURI:String
OWLObjectProperty
OWLClassExpression
OWLDataProperty
OWLDataRange
XSDDatatype
typeName:String
range
domain
superCl
subCl
default
RDB MM (extended)
/refDb
Database
dbName:String
jdbcDriver:String
connection:String
schema:String
publicTablePrefix:String
DBFunction
fName:String
Table
tName:String
View
PKey
Column
colName:String
FKey
SQLDatatype
typeName:String
/datatype
main
target
datatype
DBRef
dbName:String
dbAlias:String
isAuxDB:Boolean
isDefaultDB:Boolean
hasTempSpace:Boolean
tempTDefTable:String
viewDefTable:String
initSQLScript:String
TTableDef
tName:String
insertSQL:String
ViewDef
viewName:String
viewDBAlias:String
TableRef
tName:String
dbAlias:String[0..1]
mainTable
DBFunctionRef
fName:String
fDBAlias:String
ClassMap
instruction:{?,?Any, ?Out,?In,!No,!Sub} [*]
/src
/trg
/ref
ObjectPropertyMap
instruction:{?Dom,?Ran} [*]
DataPropertyMap
instruction:{?Dom} [*]
Reference
varName:String
/def
DataExpression
functionBody
/evalContext
TableExpression
/sqlView:String
/exprContext
exists
XSDRef
typeName:String
ColDef
colName:String
TopItemSpec
RefItem
prev
next
ClassMapRef
mark:{<s>,<t>, empty}
NavigItem
NavigLink
symbol:{->, =>}
left
right
rLink
lLink
rightC
leftC
ValueList
KeyItem
KeySequence
AggrBaseRef
mark:{<b>}
ExprItem
alias:String
NamedRef
refName:String
TableRefExpr
alias:String
FilterSpec
TopFilterSpec
RegularFilterInNavig
TopSpecificationBody
count: Integer [0..1]
percent: 1..100 [0..1]
rowNum: Integer [0..1]
OrderList
OrderItem
desc: Boolean
BooleanExpr
filterExpr
ExistsFilter
ConstantFilter
opName:{true,false}
NotFilter
cond:{NOT}
AndOrFilter
cond:{AND,OR}
BinaryFilterItem
opName:{=,<,>, <=,>=,<>,like}
UnaryFilterItem
opName:{is null, is not null}
BetweenExpr
BooleanValueExpr
DefVarRef
ClassRef
ColPrefix
text:String
FunctionDef
fName:String
isAggregate:Boolean
hasXSDRef:Boolean
/function
Param
pName:String
Parameters @TExpr, @Col
AggregateFunctionDef
/application
CustomAggregate
aggrName:String
BuiltinAggregateExpr
aggrName:{Sum, Count, Min, Max, Avg}
@@aggregate(@1, concat(@1,', ',@2))
RawAggregationExpr
AggregateExpr
URIPattern
URIItem
first
joinOp
ValueExpression
/hasXSDRef:Boolean
args
parent
child
BinaryExpr
op:{=,<,>,>=,<=, <>,+,-,*,/,div,mod}
UnaryExpr
op:{-}
SQLExpr
Constant
cValue:String
Variable
vName:String
FunctionExpr
fName:String
SQLFunction
RDB2OWLFunction
DefFunctionExpr
TableFunction
SourceDBFunction
CompoundColRef
ColumnRef
ColNameRef
colName:String
DefColName
TableColName
{ordered}
1
*
0..1
1..*
2
3