Advanced RDB-to-RDF/OWL mapping facilities in RDB2OWL

Guntars Būmans[1,*], Kārlis Čerāns[2,**]

[1] Department of Computing, University of Latvia, Raiņa bulvāris 19, Riga LV-1586, Latvia
guntars.bumans@gmail.com
[2] Institute of Mathematics and Computer Science, University of Latvia
Raiņa bulvāris 29, Riga LV-1459, Latvia
karlis.cerans@lumii.lv

Abstract. We present advanced features of RDB2OWL mapping specification language that allows expressing RDB-to-RDF/OWL mappings in a concise and human comprehensible way. The RDB2OWL mappings can be regarded as documentation of the database-to-ontology relation. The RDB2OWL language uses the OWL ontology structure as a backbone for mapping specification by placing the database link information into the annotations for ontology classes and properties. Its features include reuse of database table key information, user defined scalar and aggregate functions, table-based functions and multiclass conceptualization that is essential to keep mappings compact in case when large tables are mapped onto several classes on the table field value existence basis.

Keywords: Relational databases, RDF, OWL ontologies, mappings

1 Introduction

The use of Semantic Web technologies has grown dramatically in recent years. It has been motivated by the need for semantic information organization both on global and on enterprise scale. Various open standards and notations have been developed such as RDF[1], SPARQL 1.1 [2], OWL 2.0 [3] to serve this purpose. RDF is data formalism specifying information coding in a triples form as a graph. The schema information for the graph may be given as RDF Schema [4] or OWL ontology. Most of the data still reside in relational databases with no near future vision for massive global RDB-to-RDF triple store migration. This calls for development of technologies for relational database integration into new Semantic Web data world. For this purpose development has been directed towards RDB-to-OWL/RDF mapping specification means (languages, techniques) and machine processing of mappings to generate RDF triples from RDB data and/or to represent RDB data as RDF triples.

A number of techniques and tools have been developed for these tasks, including R_2O [5], D2RQ [6], Virtuoso RDF Views [7], DartGrid [8] as well as UltraWrap [9]

[*] Partially supported by ESF project 2009/0138/1DP/1.1.2.1.2/09/IPIA/VIAA/004.
[**] Partially supported by ERAF project 2011/0009/2DP/2.1.1.1.0/10/APIA/VIAA/112

and Triplify [10]. There is upcoming W3C standard R2RML [11] for RDB-to-RDF mapping specification, as well as a standard for direct (technical) mapping [12] 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 [13] approach, or by means of some RDF-to-RDF mapping language such as R2R [14], or some model transformation language (see e.g. [15] for an example approach).

The RDB2OWL approach we are elaborating here deals with defining mappings between source database schema and conceptual data ontology that may substantially differ from the "technical" presentation of the database schema. The existing R_2O, D2RQ, Virtuoso RDF Views and R2RML languages already cover well the low-level constructs that are essential in this context.

Within the RDB2OWL approach we focus on offering a mapping specification language [16, 17] that allows creating human readable mapping specifications and attaching those as annotations to the target ontology classes and properties. The main contribution of this paper is the description of the advanced high-level constructs in the RDB2OWL mapping specification language that allows keeping the mapping definition fully human-comprehensible also in the case of complex mapping structure.

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 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) [18, 19]. This approach benefits from SQL processing speed of modern RDBMS, therefore, the combination of a high level specification language with efficient implementation structures is achieved. As a future perspective: RDB2OWL human readable mapping language can be transformed to R2RML language to use any tool that implements it.

After presenting a brief review of basic RDB2OWL notations, we move on to explanation of advanced mapping constructions:

- multiclass conceptualization to obtain compact mapping specifications for a parent class and its subclasses mapped to the same database table;
- creation and usage of auxiliary database schema objects that can help defining the source database to ontology mapping;
- user defined and built-in RDB2OWL scalar and aggregate functions; function definition expressions can include references to source and auxiliary database tables and columns to enhance expressiveness.

We discuss both the basic and advanced RDB2OWL mapping specification language on a simple mini-University example [17], as well as Latvian medicine registries RDB-to-RDF/OWL migration case [20, 21].

2 RDB2OWL Core Language: an Overview

A RDB2OWL mapping is a relation from a source RDB schema S to target OWL ontology O, specifying the correspondence between the source database data and RDF triples "conforming" to the target ontology. The RDB schema to OWL ontology

mapping consists of individual entity-level mappings, each attached to an OWL entity (class or property) specifying how to generate RDF triples for this entity from relational DB table data. A mapping for an OWL class is called *class map*, mapping for an OWL property - *property map*. There are *datatype property maps* and *object property maps* for OWL datatype and object properties respectively.

We illustrate the basic mapping notions and syntax by mini-University example adopted from [22]. A database schema and corresponding target ontology for the example is shown in Figure 1. The basic mapping building pattern is in marking OWL classes as corresponding to relational tables, data properties to table columns and object properties to foreign key relations. The example shows, however, that even in small-sized cases the mappings are not exactly one-to-one. For example, a class *Course* that corresponds to table *XCourse* is split into subclasses on the basis of *required* column of *XCourse* table (there is a similar split for *Teacher* class). There is a class *Person* that covers *Student* and *Teacher* classes in the ontology. The property *takes* is *Student*-to-*Course* n:n relation, it corresponds to *XStudent-XRegistration-XTeacher* table pattern (the RDB model doesn't support n:n relations between tables).

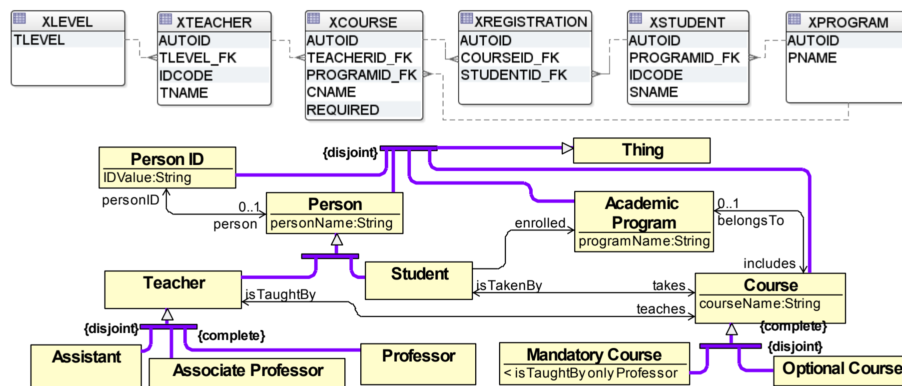

Fig. 1. A RDB schema and ontology of mini-University

Figure 2 shows mini-University ontology annotated in accordance with RDB2OWL core mapping language. A custom extension of OWLGrEd editor [23] is used for showing graphically ontology together with annotations; the *DBExpr*-annotations for classes and properties are depicted as '{DB: <annotation text>}'.

Some simple class map examples in Fig.2 are *XStudent*, *XTeacher*, *XCourse;* each of them relates an ontology class to a database table. The class maps for derived classes (e.g. the *Assistant* class) in the example include a reference to the sole class map defined at another class (e.g., *[[Teacher]]*), followed by a filter (e.g. *Level='Assistant'*). The class *PersonID* intuitively is based on *IDCode* column for both *XTeacher* and *XStudent* tables; formally this dependence is described by ascribing two named class maps (*S* and *T*) to *PersonID* class; each of these class maps specify both the table on which the *PersonID* class is based on and the way of *PersonID* class instance URI formation on the basis of the data contained in *XTeacher* or *XStudent* table row, corresponding to this instance.

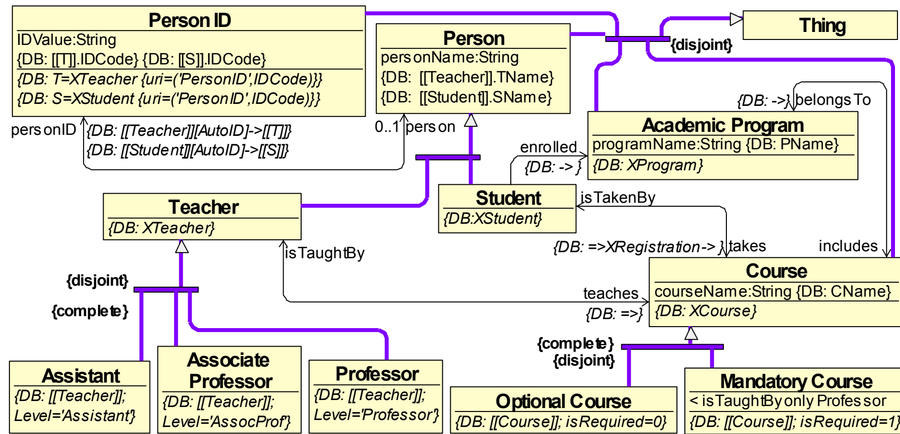

Fig. 2. Annotated mini-University ontology using RDB2OWL Core model [17]

An object property map is defined as a table expression typically involving explicitly or implicitly specified tables corresponding to the property domain and range classes. So, the property map for property *enrolled*, depicted in the diagram as ->, could be described also in a more detailed way as <*s*>-><*t*> (<*s*> and <*t*> are "reference marks" for sole class maps ascribed to the domain and range classes of the property), or as *(XStudent <s>)->(XProgram <t>)* (the referred table names made explicit), or *(XStudent <s>)[ProgramID_FK]->[AutoID](XProgram <t>)* (explicit specification of table linking columns), or *(XStudent <s>), (XProgram <t>); <s>.ProgramID_FK = <t>.AutoID* (a simple table expression form without navigation expressions). The expanded expression forms can be used instead of the simple form whenever this is convenient. Omitting the column information within the navigation link is possible whenever the navigation link corresponds to the sole foreign key to primary key relation between two tables (for the relation in a "backwards direction" the column information may be omitted, as well, while changing the navigation link symbol from -> to =>).

The navigation expressions can be chained, as in the object property map for *takes* property: =>*XRegistration*->. By inserting the implicit tables, it is expanded to (*XStudent <s>)=>XRegistration->(XCourse <t>)*.

Syntactically a table expression consists of a comma-separated list of *reference items*, optionally after semicolon followed by a filter expression. Each reference item can be a table name or a nested table expression, followed optionally by an alias. Some table expression examples are, as follows:

1. *XStudent*
2. *XRegistration R, XCourse C, XTeacher T;*
   R.CourseID_FK=C.AutoID AND C.TeacherID_FK=T.AutoID
3. *(XStudent S, XProgram P; X.ProgramID_FK=P.AutoID) SP,*
   (XRegistration R, XCourse C; R.CourseID_FK=C.AutoID) RC;
   SP.(S.AutoID)= RC.(R.StudentID_FK)

Furthermore, a reference item can be expressed as a *navigation list* e.g., *XStudent [ProgramID_FK]->[AutoID]XProgram* where *[ProgramID_FK]->[AutoID]* is a

navigation link which can be simplified to ->, as described above. Navigation links can be based also on comparison of *value expression* evaluation in respective tables, such as *(Person P1)[salary]->[salary*2] (Person P2)*, or on column list correspondence, as in: *(XRegistration R1)[CourseID_FK, StudentID_FK]->[CourseID_FK, StudentID_FK] (XRegistration R2)*.

The navigation items in table expressions may include also row filtering conditions: *XStudent:(sname='Alice')[ProgramID_FK]->[AutoID] Xprogram*

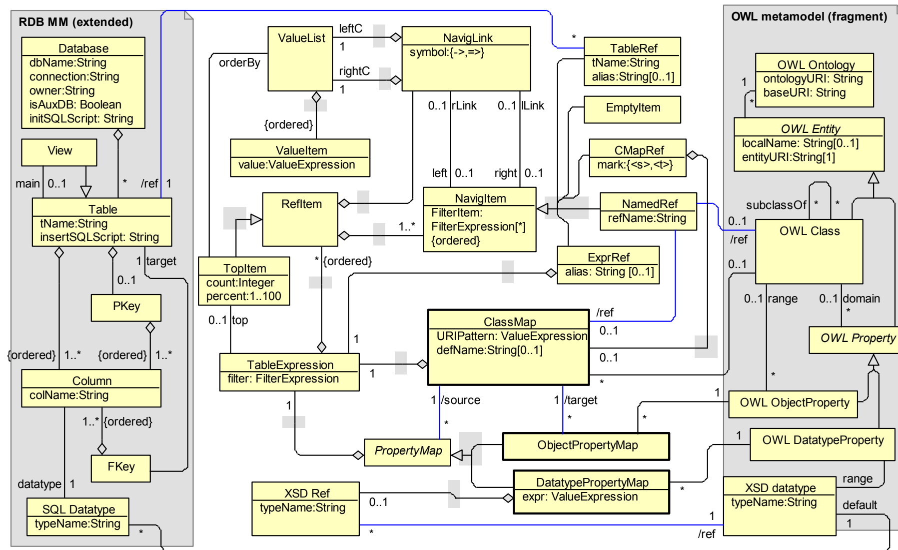

Fig. 3. RDB2OWL Core metamodel

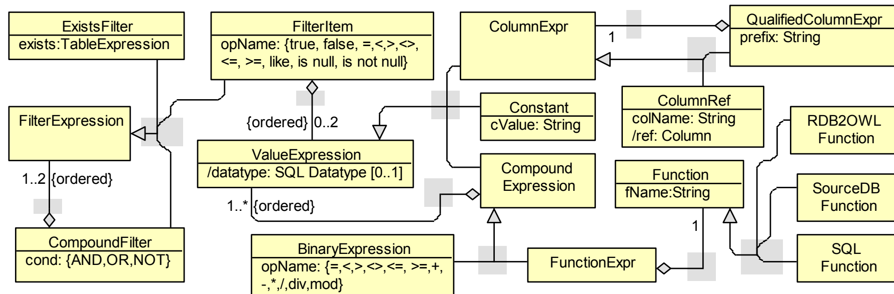

Fig. 4. Expression and filter metamodel

A *datatype property* map is described by a table expression that is required to contain (explicitly or implicitly) a single <s>-marked reference to its source class map (typically, the sole class map for the property domain class); followed by a value expression after a dot, and further on by an optional datatype specification preceded by '^^'. The following are examples for property map specification:

– *XCourse <s>->XProgram P.concat(<s>.cname, ", ", P.pname)*
– *XCourse <s>.cname*
– *<s>.cname*
– *cname*   ('*<s>*.' can be omitted)

Figure 3 summarizes the basic notions of RDB2OWL Core mapping language in a metamodel form; Figure 4 provides a supplementary expression and filter metamodel. A more detailed description of core RDB2OWL mapping constructs is in [17].

3 Advanced RDB2OWL Constructs

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 [20, 21]. In the considered example the relational database contained 3 GB data consisting of 106 tables, 1353 table columns and 3 million rows in total, as reported in [18, 19]. The corresponding ontology had 172 OWL classes, 814 OWL datatype properties and 218 OWL object properties [19].

3.1 Multiclass Conceptualization

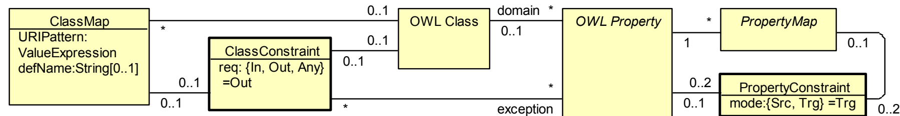

Fig. 5. Class and property constraint metamodel

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.

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.

3.2 Auxiliary Database Objects

There are cases when direct mapping between source RDB and the target ontology is not possible or requires complex expressions involving manual SQL scripts. Additional databases and its tables can be introduced for the mapping purpose. There can be multiple *Database* class instances in RDB2OWL core metamodel (see Figure 3). 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.6, 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.

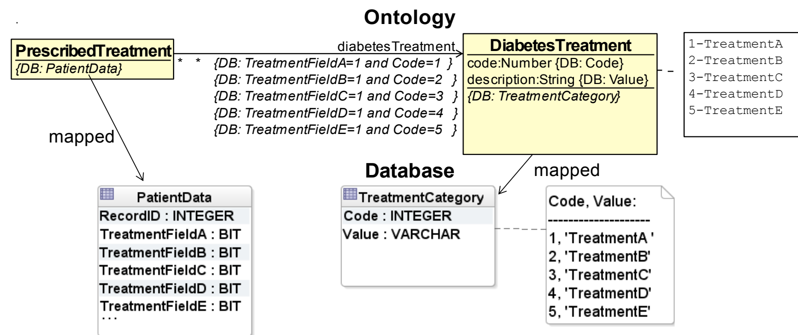

Fig. 6. Ontology and Database fragment for Diabetes Treatment modeling

3.3 RDB2OWL Functions

Possibility of function definition and use increases substantially the abstraction level of programming notation. In practical RDB2OWL mapping use cases the functions have been important e.g. to cope concisely with legacy design patterns present in the source database. A basic RDB2OWL function metamodel is shown in Figure 7.

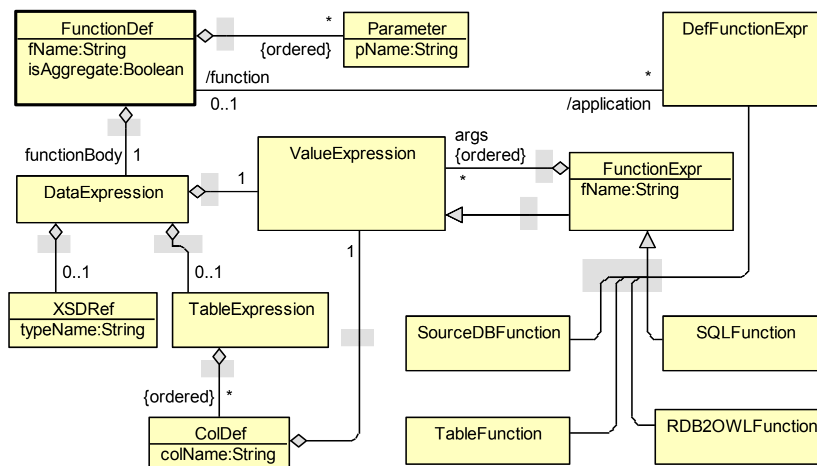

Fig. 7. RDB2OWL Function metamodel

We introduce scalar-argument as well as aggregate functions into RDB2OWL (aggregate functions are shown in Figure 8 and are described in section 3.4). 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*).

3.3.1 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 1. Built-in RDB2OWL 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.

The list of built-in functions is to be finalized within upcoming RDB2OWL reference manual.

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

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

$(A,T;filter'; colDef'_1, ...,colDef'_m).val[V_1/@X_1, ..., V_n/@X_n]$, 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 [24]), 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.

3.4 Aggregate Functions

Fig. 8. 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 1 and Fig 2), 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=(AutoID)} => XCourse.AutoID)

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>=>XRegistration->XCourse {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((=>XRegistration->XCourse {Code asc}).Code)
In this example the table expression *=>XRegistration->XCourse {Code asc}* is assigned to variable *@TExpr* and the value expression *Code* – to variable *@Col*.

3.5 Extended Mapping Example

We present an example illustrating the advanced RDB2OWL construct application. Figures 9 and 10 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.

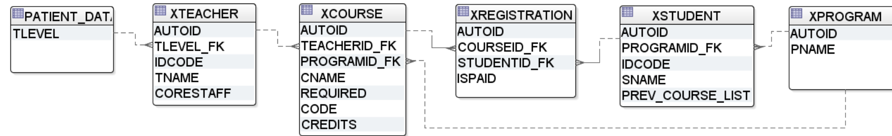

Fig. 9. An extended mini-university database schema

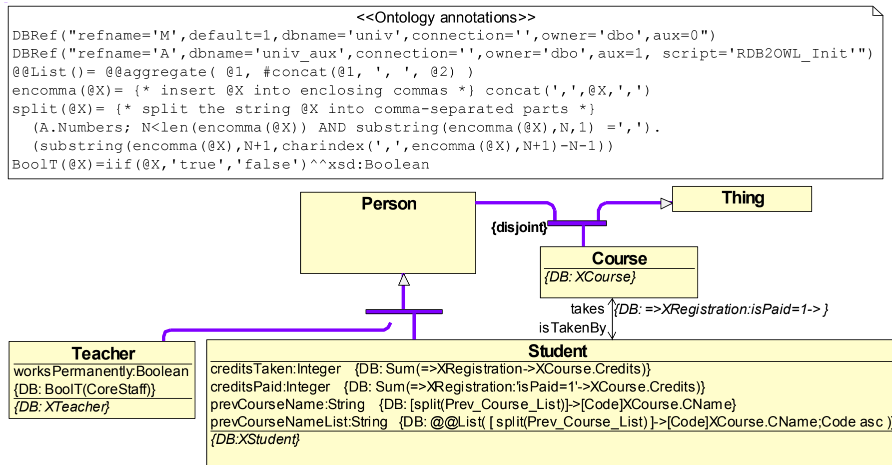

Fig. 10. Extended ontology example

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 Conclusions

We have presented advanced features of RDB2OWL approach for RDB-to-RDF/OWL mapping specification that are written as OWL ontology annotations. Convenience of mapping definitions has been illustrated by example. Power of RDB2OWL combined with visual ontology modeling tools such as OWGrEd [23] can be viable mechanism for RDB semantic re-engineering. An annotated OWL ontology with RDB mappings can be thought also as a documentation describing the technical data schema from the perspective of the conceptual model (ontology).

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 in real life case of semantic re-engineering of 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 (the aggregate functions were not needed in the defined mapping case). 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).

Another interesting venue of research would be translation of RDB2OWL mappings into more commonly used RDB-to-OWL mapping formalisms, such as D2RQ or Virtuoso RDF Views, to enable on-the-fly access to the relational databases from a RDF/SPARQL endpoint, on the basis of specified RDB2OWL mappings.

References

1. Resource Description Framework (RDF), http://www.w3.org/RDF/
2. SPARQL 1.1 Query Language, http://www.w3.org/TR/2010/WD-sparql11-query-20100601/
3. OWL 2 Web Ontology Language, Structural Specification and Functional-Style Syntax http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/
4. RDF Vocabulary Description Language 1.0: RDF Schema http://www.w3.org/TR/rdf-schema/
5. 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).
6. D2RQ Platform, http://www4.wiwiss.fu-berlin.de/bizer/D2RQ/spec/
7. C.Blakeley: RDF Views of SQL Data (Declarative SQL Schema to RDF Mapping), OpenLink Software, 2007.
8. W.Hu, Y.Qu: Discovering Simple Mappings Between Relational Database Schemas and Ontologies, In Proceedings of 6th International Semantic Web Conference (ISWC 2007), 2nd Asian Semantic Web Conference (ASWC 2007), LNCS 4825, pages 225-238, Busan, Korea, 11-15 November 2007.
9. 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)
10. 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).
11. R2RML: RDB to RDF Mapping Language, http://www.w3.org/TR/r2rml/
12. A Direct Mapping of Relational Data to RDF, http://www.w3.org/TR/2011/WD-rdb-direct-mapping-20110324/
13. Christian Perez de Laborda and Stefan Conrad: Database to Semantic Web Mapping using RDF Query Languages LNCS 4215, pp. 241-254. Springer, Heidelberg, 2006.
14. 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.
15. 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.
16. G.Būmans, K.Čerāns, RDB2OWL: Mapping Relational Databases into OWL Ontologies - a Practical Approach. Databases and information systems: Proc. of the Ninth International Baltic Conference, Baltic DB&IS 2010, Riga, Latvia, July 5-7, 2010. Riga, University of Latvia (2010), 393-408.
17. Kārlis ČERĀNS, Guntars BŪMANS: RDB2OWL: a RDB-to-RDF/OWL Mapping Specification Language, IOSPress, vol.224, 2011 (ISBN 978-1-60750-687-4) pp. 139-152.
18. 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.
19. G.Būmans, K.Čerāns, RDB2OWL: a Practical Approach for Transforming RDB Data into RDF/OWL, in Proceedings of the 6th International Conference on Semantic Systems, Graz, Austria, September 2010, ACM International Conference Proceeding Series, ISBN 9781450300148 (2010) Article No.25.
20. 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 (2008), 89-102.

21. G.Barzdins, S.Rikacovs, M.Veilande, and M.Zviedris, Ontological Re-engineering of Medical Data-bases, Proc. of the Latvian Academy of Sciences, Section B, Vol. 63 (2009), No. 4/5 (663/664), 20–30.
22. 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
23. OWLGrEd, http://owlgred.lumii.lv/
24. J.Moden. The "Numbers" or "Tally" Table: What it is and how it replaces a loop. http://www.sqlservercentral.com/articles/T-SQL/62867/