MDA: Correctness of Model Transformations. Which Models Are Schemas?

Karlis PODNIEKS
Institute of Mathematics and Computer Science, University of Latvia, 29 Raina Boulevard, Riga, LV-1459, Latvia

Abstract. How to determine, is a proposed model transformation correct, or not? In general, the answer may depend on the model semantics. Of course, a model transformation is “correct”, if we can extend it to a “correct” instance data transformation. Where should model semantics be defined? Assume, model syntax and semantics are defined in the same meta-model. Then, how to separate syntax from semantics? The answer could be the definition of model schemas proposed in the paper.

Outline

The paper is structured as follows. Section 1 discusses the correctness of model transformations that are solving the UML to RDBMS transformation problem. Section 2 introduces an extension of this problem, in which model syntax and semantics are defined in the same meta-model. Section 3 is the core of the paper – it proposes a general definition of model schemas. Section 4 applies this definition to XML-schemas. Section 5 considers the relationship between schemas and model constraints. Finally, Section 6 discusses the related work.

1. Which UML to RDBMS Transformations Are Correct?

The Object Management Group (OMG) has issued a Request for Proposal for a Query/Views/Transformations (QVT) language that would allow defining of mappings between different information models [1]. “In defining mappings from model to model, the question of correctness of the mapping arises. ... The more complex form of correctness is that of semantic correctness; does the result of transformation *mean* the same thing as the input?“ [2].

Indeed, let us consider a fragment of the example problem used by MOF QVT submitters, the so-called UML to RDBMS transformation problem ([3], Section 5.1.6). Figure 1 represents fragments of the input and output meta-models. Figure 2 represents an example input model – an interpretation of the UML meta-model.

The transformation problem is expressed as follows. The input model is an interpretation of the input meta-model. It consists of persistent and transient classes owning attributes. Attributes may be primitive (having a primitive data type), or complex (having a transient class as a type). The output model is an interpretation of the output meta-model. It consists of tables owning columns. The transformation in question must: a) Transform each persistent class into a single table. b) Transform each class attribute of a primitive type into a column of the corresponding table. c) “Drill down” class attributes of complex types to leaf-level primitive attributes; transform these primitive attributes into columns of the corresponding table.

How to determine, is a proposed transformation of this kind “correct”, or not?

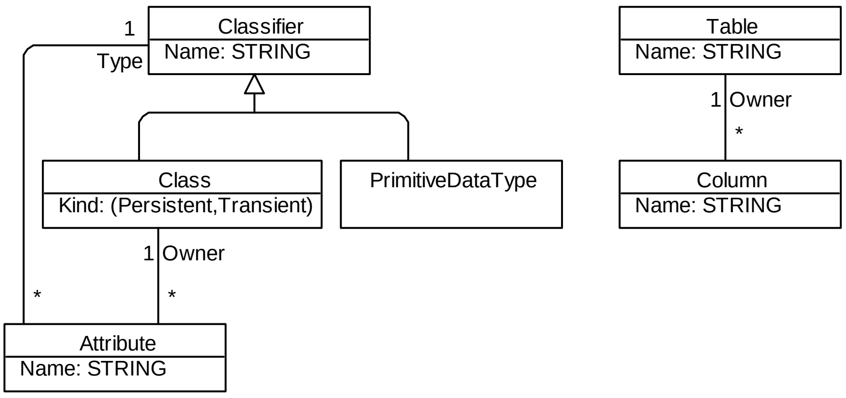

Figure 1. Fragments of UML and RDBMS meta-models

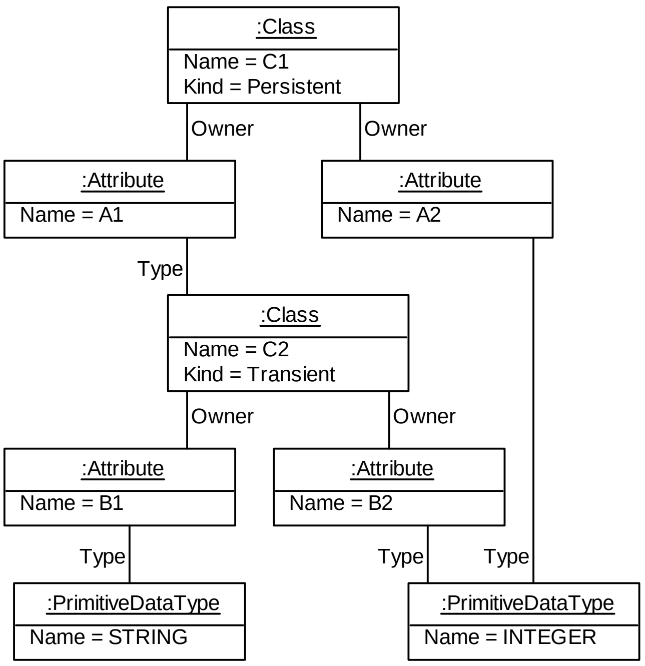

Figure 2. Example input model – an interpretation of the UML meta-model

Let us consider two different transformations that are transforming (uniformly) any UML model into a RDBMS model. These transformations will be demonstrated for the example input model represented in Figure 2. The generalization is obvious.

“Absolutely lossless” transformation. The following trivial transformation *T1* should be regarded as “absolutely lossless” – it transforms Figure 1 into Figure 3:

a) *T1* transforms a persistent class named *C1* into a table named *t_C1_Persistent*.

b) If the class *C1* owns an attribute *A1*, and the type of *A1* is a transient class *C2*, and *C2* owns an attribute *B1* of a primitive type *STRING*, then *T1* creates a column named *c_A1_C2_Transient_B1_STRING*.

c) Similarly, in all the other situations.

T1 is an “absolutely lossless” transformation, because it is reversible – **no information gets lost** during the transformation. Indeed, having an output model, generated by *T1*, we can restore all elements of the input model.

Note. Of course - with the exception of the transient classes that are not used as attribute (or sub-attribute) types in persistent classes.

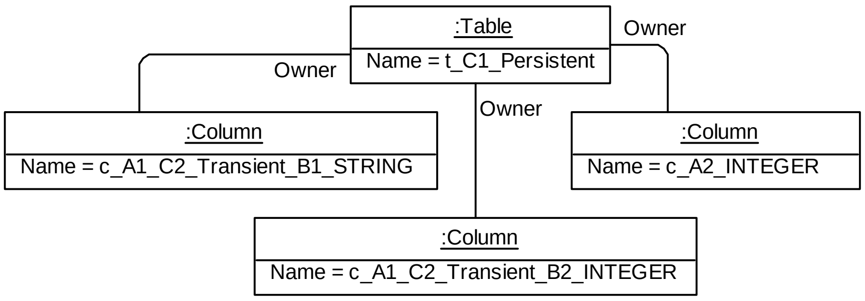

Figure 3. Example output model created by the transformation *T1*

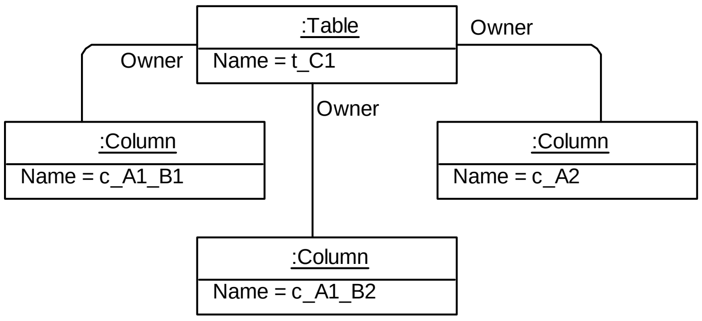

Figure 4. Example output model created by the transformation *T2*

Practical transformations do not need to be “absolutely lossless”. Of course, none of the actual MOF QVT proposals is using this “absolutely lossless” transformation *T1* (see, for example, [3]). Instead of *T1*, they are using another transformation *T2*, which transforms Figure 2 into Figure 4, and thus, differs from *T1* as follows:

b) If the class *C1* owns an attribute *A1*, and the type of *A1* is the class *C2*, and *C2* owns an attribute *B1*of a primitive type *STRING*, then *T2* creates a column named *c_A1_B1*.

When compared to the *T1*’s version of the column name *c_A1_C2_Transient_B1_STRING*, the transformation *T2*, in its version *c_A1_B1*, omits the intermediate class name and attribute (*C2_Transient*), and the primitive type name (*STRING*). Thus, *T2* is not completely reversible - the information about names of transient classes and primitive types **gets lost during the transformation**. Why should we regard this widely used *T2* as a “correct” transformation? In which sense, the result of *T2* “*means* the same thing as the input“ [2]?

The intended semantics of the UML to RDBMS transformation is as follows. We do not need this transformation by itself. We need it as a **basis for instance data (database contents) transformations**. The input UML model can be regarded as a schema of an object-oriented database, and the output RDBMS model – as a schema of a relational database. Thus, in fact, to solve the UML to RDBMS transformation problem completely, we must provide not only the model (i.e. database schema) transformation. To make it useful, we must provide also the instance data (i.e. database contents) transformation that would allow converting (without loss of information) the contents of any object-oriented database into the contents of a relational database. The solution of this problem is a well-known topic described in database textbooks for students.

And, of course, for the above small fragment of the problem (Figure 1), the database contents transformation *D2* extending the schema transformation *T2* is trivial. In the resulting database created by *D2,* we do not represent the instances of intermediate complex attributes (like as *A1* in Figure 2), and links connected to them. But, nevertheless, *D2* **is completely reversible**. Indeed, we can restore easily the contents of the input (object-oriented) database from the contents of the output (relational) database, by using, additionally, the information contained in the database schemas:

a) For each row of the table *t_C1*, create an instance of the class *C1*.

b) For a cell corresponding to the column *c_A1_B1*, and containing the value *"123"*, create (if not created before) an instance of the attribute *A1*, and link it to the corresponding *Owner* instance of *C1*, create (if not created before) the corresponding *Type* instance of the class *C2*, and create an instance of the attribute *B1* containing the value *"123"*, and link it to the corresponding *Owner* instance of *C2*.

c) Similarly, in all the other situations.

Thus, by referring to database schemas, we can restore all the input database information, missing in the output database. And thus, the pair *T2+D2* can be regarded as a lossless transformation.

Note. Of course, in the MDA context, many transformations do not need to be lossless. In MDA, transformations may lose information; they may merge parts of several models, add new information via user interfaces etc. In MDA, a model transformation is acceptable, if it performs its task.

2. Where Should Model Semantics Be Defined?

As we now see, it may happen that specifying the correctness of model transformations may be impossible, if we restrict the problem to the model syntax, and ignore **model semantics**.

In [4], after considering several model management operators, the author concludes (see his Section 3.10): "The model management operators defined in Section 3 are purely syntactic. That is, they treat models and mappings as graph structures, not as schemas that are templates for instances… Still, in most applications, to be useful, models and mappings must ultimately be regarded as templates for instances. That is, they must have semantics. Thus, there is a semantic gap between model management and applications that needs to be filled."

Mathematical theories are formalized by using the first order predicate logic, i.e. by using some first order language, the axioms of predicate logic and by assuming the necessary specific axioms of a particular theory. The generally acknowledged technique of exploring the "semantics" of formal mathematical theories is the notion of **interpretation** (see any logic textbook: sorts, their interpretation domains, interpretations of constant letters, function letters and predicate letters, standard interpretations of logical connectives and quantifiers). In computer science, finite interpretations (i.e. interpretations with finite domains) are, in general, more important than the infinite ones.

In modeling, usually, the formal aspects of models are specified by using meta-models. Meta-models are serving here as "theories of models". Hence, by applying the widely approved technique of mathematical logic, we could propose to think of **models as interpretations of their meta-models**.

Frequently, meta-models are represented as UML class diagrams (together with sets of constraints written in OCL). Of course, these diagrams (together with their constraints) can be represented as first order formal theories (if necessary, small subsets of set theory may be involved). Thus, a correct semantics of meta-model diagrams can be obtained automatically by applying the above-mentioned standard notion of interpretation.

In [5], the authors argue (at the end of their Section 5.3): "there are many reasons to avoid stating that "a model is an instance of a meta-model because its elements are instances of meta-model-elements". Indeed, the relationship between a theory and its interpretations is much more complicated than the relationship between a class and its instance objects.

Note. Sadly enough, in mathematical logic and in computer science, the term "model" has opposite meanings. In mathematical logic, "a model of a theory" is an interpretation under which all axioms of the theory become true (i.e. model is a kind of "reality" modeled in the theory). In computer science, "model" means some formal structure that can replace a fragment of reality in our reasoning about this fragment (i.e. model is a kind of "theory" modeling a fragment of reality). In such a situation, speaking of models is misleading, and to avoid this, the term "interpretation" could be used instead: an interpretation of a theory (of a meta-model) is an interpretation under which all axioms (constraints) of it become true.

In many cases, meta-models define mainly the allowed syntax of the corresponding models, and not their full semantics. Where should model semantics be defined?

As an example, let us consider the models (database schemas) corresponding to the meta-models represented in Figure 1, and represented in Figures 2, 3, 4. In Figure 4 we see the "table" *t_C1* (what's a table?), which owns three "columns" – *c_A1_B1*, *c_A1_B2*, and *c_A2* (what's a column?). Of course, we know that, in fact, each table is a collection of **rows** (not mentioned in schemas); each row consists of **cells** (not mentioned in schemas); each cell carries a value and corresponds to one of the columns (assigned to the table); and, in each row, each column (assigned to the table) is represented by at most one cell. But, of course, this knowledge cannot be derived from Figure 4, which represents (according to its meta-model of Figure 1) only the data specific to the database consisting of a single three-column table *t_C1*, and not the general semantics of relational databases.

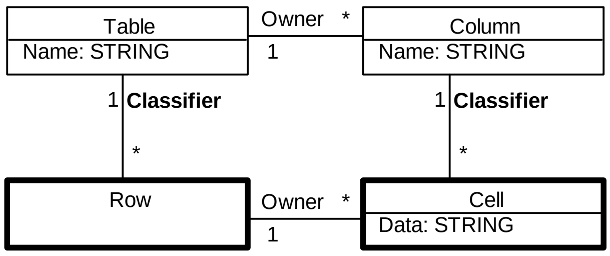

Figure 5. Fragment of RDBMS semantics meta-model (compare with Figure 1)

However, this knowledge **can be derived** from the RDBMS semantics meta-model represented in Figure 5 (with the following non-graphical constraint added: in each row of a particular table, each column – of this table - is represented by at most one cell).

Note. The above-mentioned non-graphical constraint can be stated in an OCL-like language as the following two statements:

Row is *Owner* of *Cell* --> *Row.Classifier* is *Owner* of *Cell.Classifier*

CellA.Owner = *CellB.Owner* --> *CellA.Classifier* <> *CellB.Classifier*

The first statement may be put alternatively as a diagram commutativity condition:

Cell.Owner.Classifier = *Cell.Classifier.Owner.*

By applying the standard notion of interpretation to Figure 5 we obtain, in fact, "two in one" – the database schema (i.e. table names with column names assigned to them), together with the database contents (i.e. cell values arranged in rows, columns and tables). This corresponds very well to the situation in the popular RDBMSs, where database schema is regarded as "internals" of each database ("database definition"). For example, in

SQL we do not use CREATE SCHEMA statements; instead, we are using CREATE TABLE statements (this allows defining of tables and their relationships "on the fly").

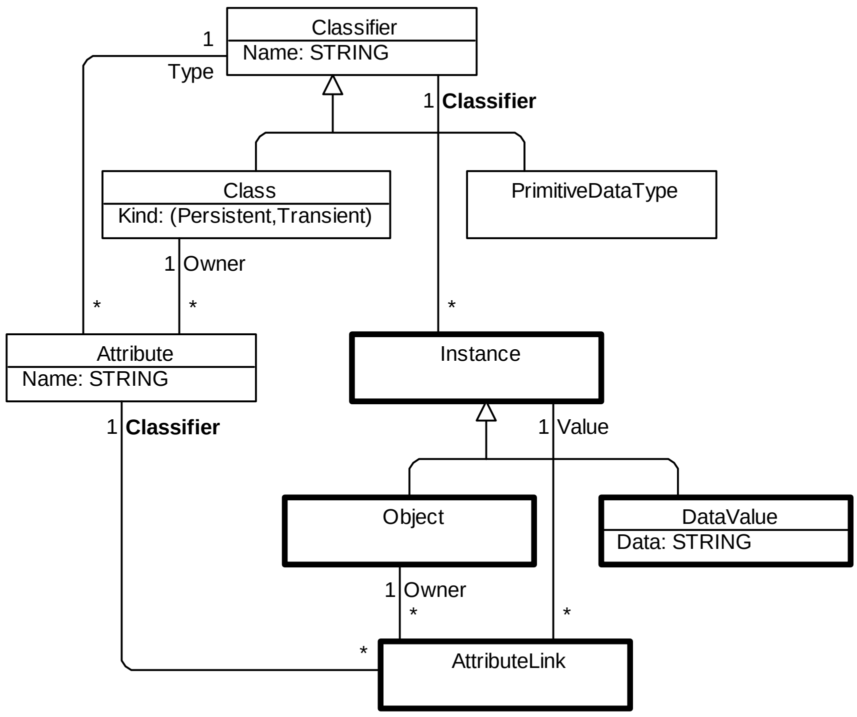

Figure 6. Fragment of UML semantics meta-model (compare with Figure 1)

Similarly, we may follow the official UML semantics definition [6], and, to capture the intended class diagram semantics missing in the UML fragment meta-model of Figure 1, extend it as represented in Figure 6 – with the following two constraints added: a) in each object of a particular class, each attribute – of this class - is represented by at most one attribute link, b) each attribute link is linked to its value – an object, or a data value, depending on the type of the corresponding attribute. After this, by applying the standard notion of interpretation, we are forced to have, again, "two in one" – classes together with their instance objects.

Note. The above-mentioned non-graphical constraints can be stated in an OCL-like language as follows:

Object is *Owner* of *AttributeLink* --> *Object.Classifier* is *Owner* of *AttributeLink.Classifier*

AttributeLinkA. Owner = *AttributeLinkB.Owner* -->
AttributeLinkA.Classifier <> *AttributeLinkB.Classifier*

Instance is *Value* of *AttributeLink* --> *Instance.Classifier* is *Type* of *AttributeLink.Classifier*

Object.Classifier is *Class*

DataValue.Classifier is *PrimitiveDataType*

The first and the third of these constraints may be put, alternatively, as diagram commutativity conditions:

AttributeLink.Owner.Classifier = *AttributeLink.Classifier.Owner*

AttributeLink.Value.Classifier = *AttributeLink.Classifier.Type*

If OMG, in its Request for Proposal for QVT language [1], would have used the Figure 5, 6 style meta-models (instead of the Figure 1 style ones), then the QVT partners would be forced to demonstrate that their proposed languages are good enough for simultaneous transformations of models and instance data.

Thus, once again, where should model semantics be defined? Now, we see a possible solution: we may **define model semantics directly in the meta-model**.

3. Which Models Are Schemas?

Is there a systematic way allowing to separate, in a meta-model, the "data" elements (boxes in bold) from the "schema" elements (regular boxes)?

Instances of "schema" elements (in Figures 5, 6 - *Table, Column, Class, Attribute, PrimitiveDataType*) are present in databases "by schema". For example, any relational database contains exactly those tables that are listed in its schema, and each table contains exactly the columns listed for it in the schema. Collections of "data" element instances, on the contrary, may differ in different databases having a common schema. For example, two common-schema relational databases containing a table named *Customers*, may contain different collections of rows of this table.

Under which conditions, a part of a meta-model could be regarded - meaningfully - as a "schema" for the rest of it?

From the above two examples the following two theses can be derived.

Thesis 1. Schema represents only the information specific to a particular model, and not the general semantics of the model-type to which it belongs.

Thesis 2. Schema defines a classification of data elements that conforms to the associations existing between these elements.

Let us try justifying Thesis 2. Simultaneously, its precise meaning will be elaborated (see Definition 1 below). Let us assume that our meta-models are defined by means of UML class diagrams with non-multiple generalizations and binary associations only.

First, the following restriction seems to be reasonable: each data element should be "named in the schema" or, more precisely, each data element should have a mandatory many-to-one or one-to-one association with some unique schema element. For data element instances, such an association defines a kind of **classification**. Thus, let us always call this distinguished association *Classifier*. For example, in Figure 5, *rows* (data element instances) are classified by *tables* (schema element instances), and *cells* (data element instances) are classified by *columns* (schema element instances). If some data element would not be linked in such a way to a schema element, then, in which sense could we speak about a "schema"?

In Figure 6, a somewhat more complicated situation appears: the data element class *Instance* consists of two subclasses – *Object* and *DataValue*. At the schema level, this triple is represented by the *Classifier* class consisting of two subclasses – *Class* and *PrimitiveDataType*. Both of these "triades" conform to the classification defined by the *Classifier* association in the following sense: *Object.Classifier* is always *Class*, and *DataValue.Classifier* is always *PrimitiveDataType*.

Thus, we have arrived at the following condition for schemas:

Condition 1 (the simplest case, Figure 5). Each data element has a mandatory many-to-one or one-to-one association (called *Classifier*) with some unique schema element (i.e. different data elements are classified by different schema elements).

Condition 1 (the full version, Figure 6). Subclasses of schema elements are schema elements. Subclasses of data elements are data elements. In the generalization hierarchy, each top-level data element has a mandatory many-to-one or one-to-one association (called *Classifier*) with some unique schema element (i.e. different data elements are classified by different schema elements). The data element generalization hierarchy is mapped into the schema element generalization hierarchy in the following sense: a) *Data_Element.Classifier* is always *Data.Element.Map*; b) If *Data_Element_1* is a subclass of *Data_Element_2,* then *Data_Element_1.Map* is a subclass of *Data_Element_2.Map*.

The next step: **associations** connecting data elements also should be represented in the schema. For example, in Figure 5, the association *Owner* connects data elements *Cell* and *Row*. The association *Owner* between *Column* and *Table,* in fact, represents this *Cell-Row*-association in the schema part of Figure 5 - because the following constraint holds (see above):

Row is *Owner* of *Cell* --> *Row.Classifier* is *Owner* of *Cell.Classifier*

Indeed, on the one hand, each cell is owned by some row, which is classified by some table. On the other hand, each cell is classified by some column, which is owned **by the same table** that classifies the row.

Thus, we have arrived at the following

Condition 2 (Figure 7). For each *Data_Association* there is a unique *Schema_Association,* to which it conforms in the following sense:

Data_Element_1 is in *Data_Association* with *Data_Element_2* -->
Data_Element_1.Classifier is in *Schema_Association* with *Data_Element_2.Classifier*

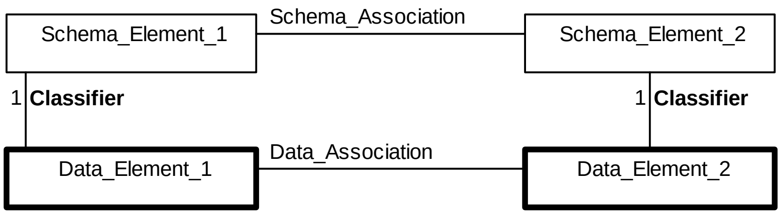

Figure 7. Representing a data association in a schema

Here, *Data_Association* conforms to the classifications of *Data_Element_1*-s and *Data_Element_2*-s: if *x* is associated with *y* via *Data_Association*, then the classifiers of *x* and *y* must be associated via *Schema_Association* (but not necessarily conversely!).

Some more specific situations are possible:

a) *x* and *y* are associated via *Data_Association*, **iff** the classifiers of *x* and *y* are associated via *Schema_Association*. Then, in fact, *Data_Association* is completely derivable from *Schema_Association*. Hence, such a *Data_Association* can be removed from the meta-model without loss of information.

b) *Data_Association* and *Schema_Association* both are many-to-one. Then Condition 2 may be put equivalently as a diagram commutativity condition:

Data_element_1.Data_Association.Classifier =
Data_element_1.Classifier. Schema_Association

c) **"Table"**. Both associations are many-to-one, and, additionally, *x1.Data_Association = x2.Data_Association* --> x1.*Classifier <> x2.Classifier*. Then, Figure 7 is isomorphic to Figure 5, i.e. we can regard instances of *Schema_Element2* as tables ("table views"), consisting of columns named by instances of *Schema_Element1*. Instances of *Data element_2* represent rows, and instances of *Data_Element_1* – cells.

Now, what about the associations, connecting data elements and schema elements, other than the *Classifier* of Condition 1? First of all, we must reject - as "non-schematic" - the associations that are connecting one instance of a data element with several instances of the same schema element. Thus, it remains to consider only many-to-one and one-to-one associations of data elements with schema elements (as *DS_Association* in Figure 8). In fact, each such *DS_Association* defines a different classification of instances of *Data_Element_1* (than the one defined by the *Classifier* association).

If *DS_Association* conforms to the "canonical" classification of *Data_Element_1,* i.e. if there is a schema association *S_Association* such that

Data_Element_1.DS_Association = *Data_Element_1.Classifier.S_Association*,

then, in fact, *DS_Association* is completely derivable from *S_Association*. Such a *DS_Association* can be removed from the meta-model without loss of information.

If, on the contrary, *DS_Association* does not conform to the “canonical” classification, then we may “refine” this classification by introducing a new *Schema_Element_12* consisting of pairs (*Schema_Element_1*, *Schema_Element_2*) and by re-defining the *Classifier* association as follows (Figure 8):

(*Schema_Element_1, Schema_Element_2*) classifies *Data_Element_1,* iff
Schema_Element_1 classifies *Data_Element_1* (in the old sense) &
Data_Element_1 is in *DS_Association* with *Schema_Element_2*.

After this, *DS_Association* becomes completely derivable from the new *Classifier* association and *S_Association* defined as follows (Figure 8):

(*Schema_Element_1, Schema_Element_2*)
is in *S_Association* with *Schema_Element_2.*

And, as such, we can remove *DS_Association* from the meta-model.

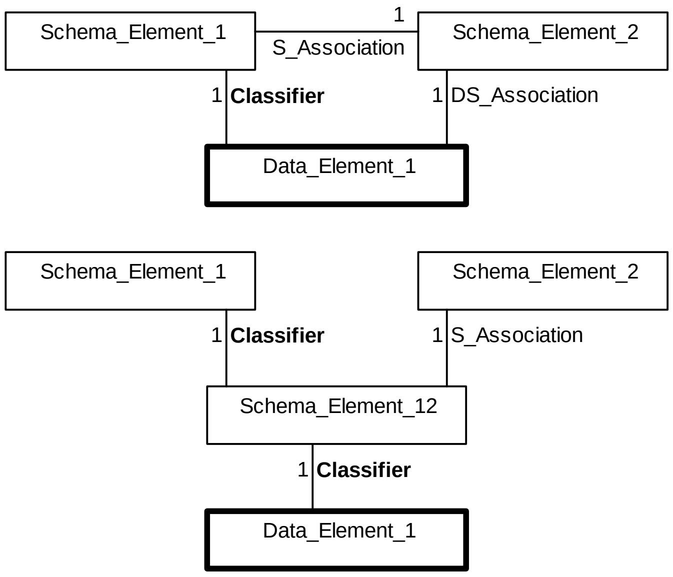

Figure 8. Representing a data-schema association in a schema

Thus, we have arrived at the following

Condition 3. *Classifier* is the only kind of associations between data elements and schema elements.

Now, we can formalize the notion of schemas:

Definition 1. Assume a meta-model that is defined by means of a UML class diagram with non-multiple generalizations and binary associations only. Assume, its classes are divided in two disjoint subsets - schema elements and data elements in such a way that the above Conditions 1, 2, 3 hold. Then, let us say that this **meta-model defines schemas**.

Note. For some of the data associations, the above-mentioned stronger “Table” version of Condition 2 may be appropriate.

Definition 1 conforms to Thesis 2. Indeed, by this definition, schemas are classifications of data instances that conform to associations existing between these elements. Thesis 2 can be used as a guideline when trying to define schemas in new situations (for an example, see Section 4 below).

Problem A. Does Definition 1 conform to Thesis 1?

Definition 2a. Assume a meta-model that defines schemas. **Model** (**database**) is an interpretation (in the sense of predicate logic) of the meta-model (i.e. an interpretation that includes both the schema part and the data part).

Definition 2b. Assume a meta-model that defines schemas. **Model schema** (**database schema**) is an interpretation of the schema part of the meta-model (i.e. each model includes its schema).

Now, in this context, let us consider model transformations. If two meta-models define schemas, then we may consider two kinds of transformations.

Definition 3a. Assume two meta-models MMA and MMB that define schemas. **Model transformation** is an algorithm transforming each MMA-model into an MMB-model in such a way that the schema part is transformed into the schema part, and the data part – into the data part.

Definition 3b. Assume two meta-models MMA and MMB that define schemas. **Schema transformation** is an algorithm transforming the schema part of each MMA-model into the schema part of an MMB-model.

To be considered as lossless, model transformations do not need to be reversible with respect to the schema part of input models. This is not necessary, because schema element instances belong to the input model "by schema", i.e. they can be restored from the meta-data - from the input schema. Thus, we can propose the following uniform definitions of lossless transformations.

Definition 4a. Assume two meta-models MMA and MMB that define schemas. Let us consider a **model transformation** D that converts any MMA-model into an MMB-model. Then D is called **lossless**, iff, there is a reverse transformation (algorithm) that restores, from any of the results of D (i.e., MMB-models), the entire **data part** of the corresponding input MMA-model.

Definition 4b. A **schema transformation** is called **lossless**, iff it can be extended to a lossless model transformation.

Both of the UML to RDBMS schema transformations considered above (the "theoretical" transformation *T1* and the "practical" transformation *T2*) can be extended to lossless model transformations (see Section 1). Thus, according to the Definition 4b, both schema transformations are lossless.

Problem B. Which reversible schema transformations (like as the above "theoretical" transformation *T1*) can be extended to lossless model transformations?

Problem C. How complicated is the task of detecting, does a lossless model transformation exist for two model schemas, or not?

4. XML-Schemas

In order to verify the above concept of schemas, let us consider XML and XML-schemas [7]. The bottom part of Figure 9 represents a simplified meta-model of "unconstrained XML", where tagged elements may be mixed up freely without any typing.

If we wish to obtain here a kind of schemas, then, according to Thesis 2, we must introduce some **classification** of XML elements. Of course, the solution is obvious: **XML-tags** define the natural XML-element classification. In this way we obtain a simplified XML-schema meta-model represented in Figure 9 - with the following non-graphical constraints added:

"For types: **no loops** are allowed via *Parent* association."

XML_Record_Type is *Parent* of *XML_Element_Type* <-->
XML_Element_Type is *Field* of *XML_Record_Type*

XML_List_Type is *Parent* of *XML_Element_Type* <-->
XML_Element_Type is *Member* of *XML_List_Type*

XML_ElementA is *Parent* of *XML_ElementB* -->
XML_ElementA.Classifier is *Parent* of *XML_ElementB.Classifier*
XML_Atom.Classifier is *XML_Atom_Type*
XML_Sequence.Classifier is *XML_ Sequence_Type*

This schema constrains XML-documents: now, an element cannot contain arbitrary sub-elements, it may contain only sub-elements of specific types pre-scribed by the schema.

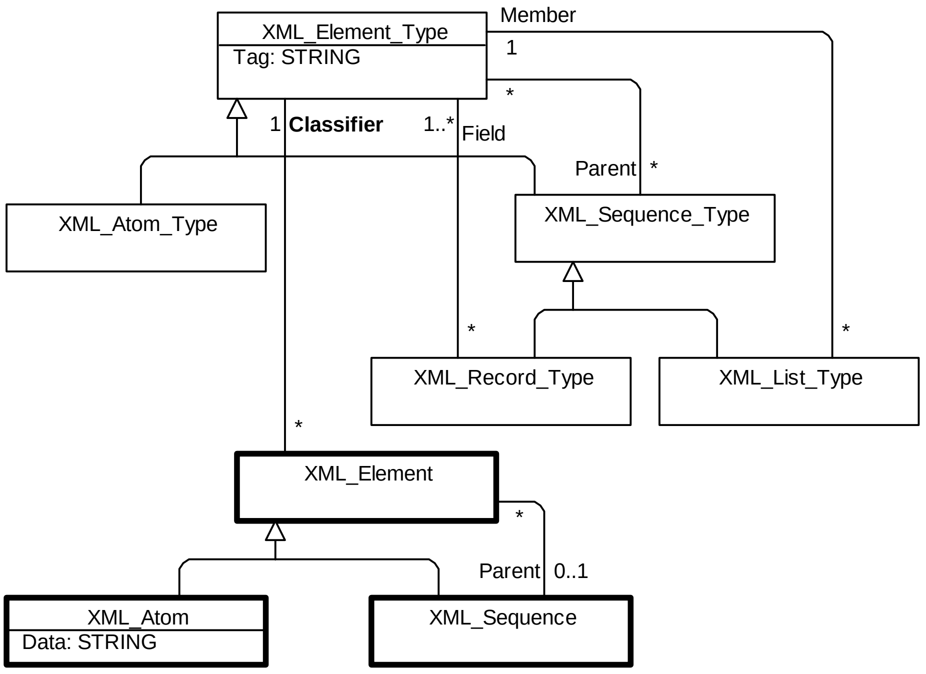

Figure 9. Simplified meta-model of XML-schemas

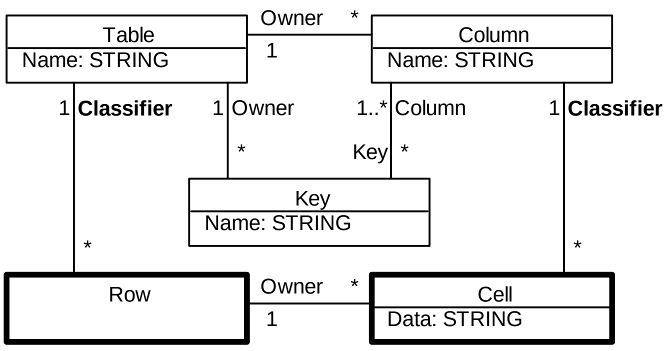

Figure 10. Extended fragment of RDBMS semantics meta-model

5. Constraints

As an example, let us consider key-constraints in relational databases. A table may possess zero or more keys, each consisting of one or more columns of this table. The corresponding meta-model is represented in Figure 10 - with the following commutativity constraint added: *Key.Column.Owner = Key.Owner.*

The meaning of the key-constraint defined by *Key* can be expressed as follows: in a table, if the cell data of two rows coincide for all columns that belong to *Key*, then these rows are equal. Or, expressed in an OCL-like language:

For All *Table, RowA, RowB*:
If *RowA.Owner = Table* & *RowB.Owner = Table* & *KeysAreEqual*(*RowA*, *RowB*)
Then *RowA = RowB,*

where *KeysAreEqual*(*RowA*, *RowB*) is the following expression:

For All *Column*, *CellA*, *CellB*:
If *Column.Key* = *Table* & *CellA.Owner* = *RowA* & *CellB.Owner* = *RowB* &
CellA.Classifier = *Column* & *CellB.Classifier* = *Column*
Then *CellA.Data* = *CellB.Data*

Thus, as a schema element, *Key* can serve only as an "enumerator" of constraints of a specific kind. The definition of the **meaning** of these constraints involves more than schema – it involves data elements and their *Data* attributes. This is not surprising (see Thesis 1): relational schemas do not define the general semantics of relational databases. Each schema represents only the information specific to a particular database.

Note. In a similar way, arbitrary functional dependencies can be specified.

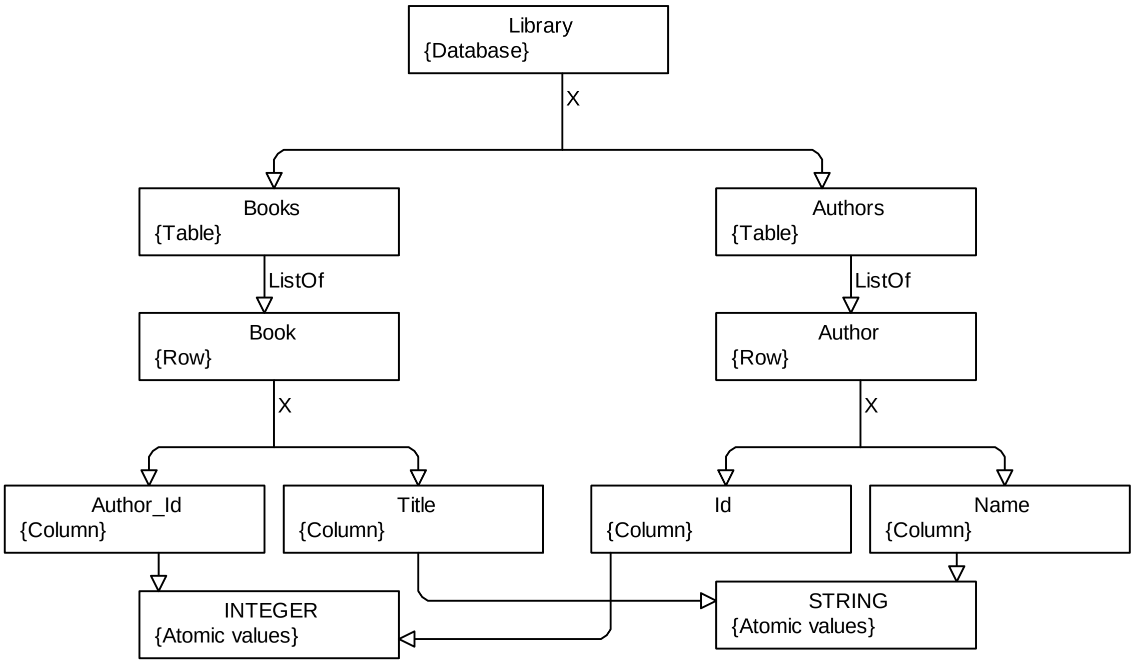

Figure 11. Example database schema according to [12]

6. Related Work

About the significance of diagram commutativity in modeling semantics – see [8].

In [9] an elegant theory of "graph schemas" for unstructured data is developed. An unstructured set of data may conform to several schemas, each of which, in its way, constrains data, thus allowing for query optimization. Despite the different setting, in their Section 5 the authors arrive at a version of the above Thesis 2: "Nodes in a schema have the potential to classify nodes in a database".

After the first versions of [10, 11], in [12] an extremely general algebraic definition of database schemas (called "abstract schemas") is proposed. To explain the basic idea, let us define a schema for a relational database *Library* consisting of two tables *Books* and *Authors* (S means "sort", see also Figure 11):

S(*Library*) = *Books* X *Authors*;
S(*Books*) = ListOf(*Book*); S(*Authors*) = ListOf(*Author*);
S(*Book*) = *Author_Id* X *Title*; S(*Author*) = *Id* X *Name*;
S(*Author_Id*) = INTEGER; S(*Title*) = STRING;
S(*Id*) = INTEGER; S(*Name*) = STRING;

Each named database element is defined here as a sort (i.e. domain) of allowed values. The last four elements are atomic, the other ones are complex, and their domains are defined by using a fixed set of type constructors (ListOf constructs lists of values, X – records of values). A database is defined then as any value of the sort *Library*.

In general, a database schema of this kind can be defined as an acyclic oriented graph of the kind represented in Figure 11 (single root, atomic sorts as leafs, accessible nodes only).

This notion of schema conforms to the above Thesis 1: schema represents only the data specific to the *Library* databases, and not the general semantics of “algebraic” databases (i.e. schemas do not define the meaning of INTEGER, STRING, ListOf and X). However, if we would restrict our databases to the relational databases only, then mentioning **rows** in the schema would become obsolete: **all** relational tables consist of rows and cells.

How about Thesis 2? Of course, sorts define a classification of data instances. Since the data model of “abstract schemas” does not include associations (they are implemented by using primary and foreign keys and the corresponding constraints), this is enough to conclude that the notion of “abstract schemas” conforms to Thesis 2.

Acknowledgements

I’m grateful for valuable discussions provided by my colleagues, especially Janis Barzdins, Audris Kalnins and Martins Opmanis. Science Council of Latvia has funded the work reported in this paper under project Nr.02-0002. Pictures were prepared by using the graphical business-modeling tool GRADE (www.infologistik.com, www.gradetools.com).

References

[1] OMG Document ad/02-04-10 (MOF 2.0 Query / Views / Transformations RFP). Available at www.omg.org.

[2] Gerber A., Lawley M., Raymond K., Steel J., Wood A. Transformation: The Missing Link of MDA. In: Proceedings of Graph Transformation: First International Conference (ICGT 2002), October 7-12, 2002, Barcelona, Spain, Lecture Notes in Computer Science, vol. 2505, Springer-Verlag, 2002, pp. 90-105.

[3] QVT Partners. Initial Submission for MOF 2.0 Query / Views / Transformations RFP. Version 1.0 (2003.03.03). Available at qvtp.org.

[4] Bernstein Ph. A. Applying Model Management to Classical Meta-Data Problems. In: Proceedings of the Conference on Innovative Database Research (CIDR), 2003, pp. 209-220.

[5] Bezivin J., Gerbe O. Towards a Precise Definition of the OMG/MDA Framework. In: ASE’01, Automated Software Engineering, San Diego, USA, November 26-29, 2001. Available online.

[6] OMG Document - formal/03-03-09 (UML 1.5 chapter 2 - UML Semantics). Available at www.omg.org.

[7] W3C Recommendation - XML Schema Part 1: Structures, May 2, 2001. Available at www.w3c.org.

[8] Johnson M., Dampney C. N. G. On the Value of Commutative Diagrams in Information Modelling. In: Proceedings of 3rd International Conference on Algebraic Methodology and Software Technology (AMAST’93), Springer Workshops in Computing, 1993, pp.45-58.

[9] Buneman P., Davidson S. B., Fernandez M. F., Suciu D. Adding Structure to Unstructured Data. In: Proceedings of 6th International Conference on Database Theory (ICDT'97), Lecture Notes in Computer Science, vol. 1186, Springer-Verlag, 1997, pp. 336-350.

[10] Diskin Z. Abstract Metamodeling, I: How to Reason about Meta- and Metamodeling in a Formal Way. In: Proceedings of the 8th OOPSLA Workshop on Behavioral Semantics, Denver, USA, November, 1999.

[11] Alagic S., Bernstein Ph. A. A Model Theory for Generic Schema Management. In: Proceedings of International Workshop on Database Programming Languages (DBPL '01), Lecture Notes in Computer Science, vol. 2397, 2002, pp.228-246.

[12] Goguen J. A. Data, Schema and Ontology Integration. In: Workshop on Combination of Logics: Theory and Applications (CombLog’04), Lisbon, Portugal, July 28-30, 2004. Available online.