Model Transformation by Graph Transformation: A Comparative Study

Gabriele Taentzer[1], Karsten Ehrig[1], Esther Guerra[2], Juan de Lara[3], Laszlo Lengyel[4], Tihamer Levendovszky[4], Ulrike Prange[1], Daniel Varro[4], and Szilvia Varro-Gyapay[4]

[1] Technische Universität Berlin, Germany
{karstene,gabi,ullip}@cs.tu-berlin.de
[2] Universidad Carlos III de Madrid, Spain
eguerra@inf.uc3m.es
[3] Universidad Autonoma de Madrid, Spain
jdelara@uam.es
[4] Budapest University of Technology and Economics, Hungary
{varro,gyapay}@mit.bme.hu
{lengyel,tihamer}@aut.bme.hu

Abstract. Graph transformation has been widely used for expressing model transformations. Especially transformations of visual models can be naturally formulated by graph transformations, since graphs are well suited to describe the underlying structures of models. Based on a common sample model transformation, four different model transformation approaches are presented which all perform graph transformations. At first, a basic solution is presented and crucial points of model transformations are indicated. Subsequent solutions focus mainly on the indicated problems. Finally, a first comparison of the chosen approaches to model transformation is presented where the main ingredients of each approach are summarized.

1 Introduction

Raising the abstraction level from textual programming languages to visual modeling languages, model transformation techniques and tools have become more focused recently. Model transformation problems can be formulated as graph transformation problems, thus, a variety of tools choose this technique as the underlying mechanism for the transformation engine. This paper aims at comparing four approaches to model transformation that apply graph transformation techniques for model transformation. These can be characterized by their tool support. The presented tools are AGG [1], AToM^3 [15], VIATRA2 [25] and VMTS [26] [16].

This paper reports on an activity which was started by the authors within the RTN "Syntactic and Semantic Integration of Visual Modelling Techniques" (SegraVis). Since the underlying concepts of these four tools are all based on graph transformation on one hand, and the tools are already used for model

transformations on the other hand, it was interesting to start a comparative study along a common case study. The tools and their underlying approaches considered within this paper, are shortly introduced within the rest of the introduction and presented in more detail, including their application to a sample model transformation, in the following sections.

AGG is a development environment for attributed graph transformation systems supporting an algebraic approach to graph transformation. It aims at specifying and rapid prototyping applications with complex, graph structured data. Since graph transformation can be applied on very different levels of abstraction, it can be non-attributed, attributed by simple computations or by complex processes, depending on the abstraction level. To reflect this wide application area for attributed graph transformation, we decided to attribute AGG graphs by Java objects. AGG supports typed graph transformations including type inheritance and multiplicities. Rule application can contain non-deterministic choice of rules which may be controlled by rule layers. Due to its formal foundation, AGG offers validation support being consistency checking of graphs and graph transformation systems according to graph constraints, critical pair analysis to find conflicts between rules and checking of termination criteria for graph transformation systems.

AToM^3 (A Tool for Multi-formalism and Meta-Modelling) [15] is a tool for the design of Domain Specific Visual Languages. It allows defining the abstract and concrete syntax of the Visual Language by means of meta-modelling and expressing model manipulation by means of graph transformation [9]. With the meta-model information, AToM^3 generates a customized modelling environment for the described language. Recently, AToM^3 has been extended with functionalities to generate environments for Multi-View Visual Languages (such as UML) and triple graph grammars [21]. The latter is useful to express the evolution of two different models, related by an intermediate one. Both new characteristics will be used for the proposed example.

VIATRA2 is an Eclipse-based general-purpose model transformation engineering (transware) framework that will support the entire life-cycle for the specification, design, execution, validation and maintenance of transformations within and between various modelling languages and domains. Using efficient importers and exporters, VIATRA2 is able to cooperate with an arbitrary external system, and execute the transformation with a native transformation model (plug-in), which is generated by VIATRA2. Its rule specification language combines the graph transformation and abstract state machines into a single paradigm. Essentially, elementary transformation steps are captured by graph transformation rules (using a rich graph pattern concept) while complex transformations are assembled from these basic steps by using abstract state machine rules as control flow specification. Furthermore, model constraints are also captured by the same graph pattern concept.

The Visual Modeling and Transformation System (VMTS) is a general purpose meta-modelling and transformation environment. VMTS is a highly configurable environment offering capabilities for specifying visual languages applying

meta-modelling techniques. VMTS uses the instantiation relationship residing between the M0 and M1 layers in the MOF standard [20], namely, the one between the UML class diagram and object diagram. The VMTS Presentation Framework (VPF) facilitates a means of rapid development for plug-ins as a customized presentation of the concrete syntax of the models. VMTS defines the model constraints in terms of OCL constraints placed in the meta-model. Since the rules in VMTS are specified by meta-model elements of the input and the output models, the transformation constraints are also expressed in OCL. VMTS has an automated support for preserving, guaranteeing, and validating constraints. The crosscutting concerns are handled with aspect-oriented techniques. The VMTS control flow is defined by stereotyped activity diagrams.

The comparison presented in this paper takes the following considerations in account: (i) We make a general comparison of the features provided by the tools with respect to the input and output models, transformation methods and other services. (ii) Based on the problems induced by the case study, we compare the solutions of the different approaches. An explicit classification criteria is developed in Section 7 to underpin an expressive comparison method which is easy to follow.

The rest of the paper is organized as follows: Section 2 describes our interpretation of the common case study specified in [4]. Section 3 contributes the AGG approach, Section 4 is devoted to the AToM^3 method, while Section 5 and Section 6 elaborate on the model transformation techniques offered by ViaTra and VMTS, respectively. A comparison is described in Section 7 concluding with future work.

2 A Sample Model Transformation: Main Requirements

As common case study we consider a model transformation from class diagrams to relational data base models. The main requirements for this model transformation are given in [4]. In this section, we recall the main requirements given in [4] and amend them with our interpretation of open issues.

Classes can be indicated as persistent or non-persistent. A persistent class is mapped to a table and all its attributes or associations to columns in this table. If the type of an attribute or association is another persistent class, a foreign key to the corresponding table is established.

If class hierarchies are transformed, only the topmost classes are mapped to tables. Additional attributes and associations of subclasses result in additional columns of the top-most classes.

Non-persistent classes are not mapped to tables, however, one of the main requirements for the transformation considered is to preserve all the information in the class diagram. That means attributes and associations of non-persistent classes are distributed over those tables stemming from persistent classes which access non-persistent classes.

In general, the transformation of attributes with classes as types can be handled in the same way as associations. Thus, the use of attributes could be restricted to values of primitive data types.

Summarizing the model transformation, the following subtasks have to be performed:

1. creation of a table for each persistent class
2. creation of columns for each attribute (association) of a persistent class
3. computation of transitive closure of class inheritance
4. transformation of attributes and associations of subclasses to columns of the topmost class' table
5. transformation of attributes and associations of connected non-persistent classes to columns of referring persistent classes' tables
6. deletion of helper structure (optional)
7. deletion of source model (optional)

Example. The following example is used to explain this principle in more detail. Figure 1 shows a sample class diagram as instance of a suitable meta-model. While classes *Person* and *Phone* are persistent, class *Address* is not. Moreover, class *Customer* inherits from class *Person*.

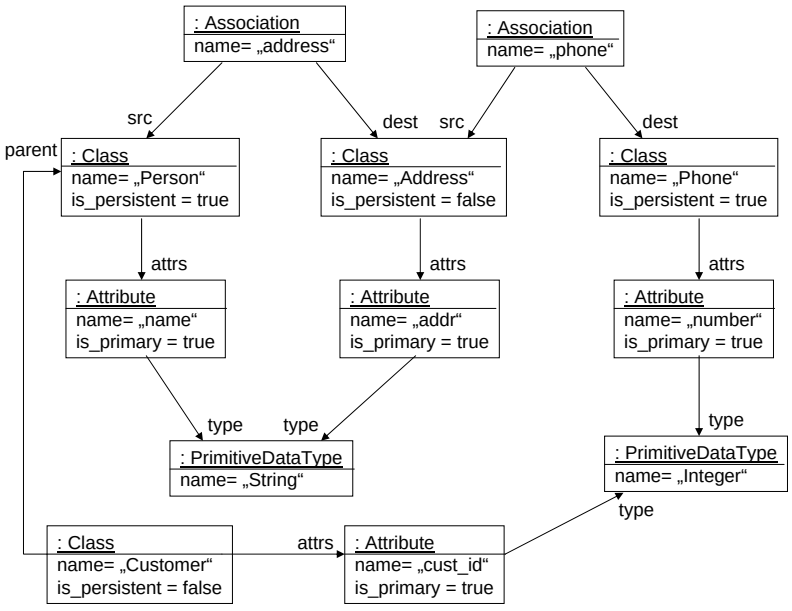

Fig. 1. Sample class diagram as meta-model instance

This class structure is transformed into the relational data base model in Figure 2 consisting of two tables, one for *Person* and one for *Phone*. The asso-

ciations and the primary attribute of class *Address* are mapped to columns of table *Person*. Furthermore, the primary attribute of class *Customer* is mapped to a column of table *Person*.

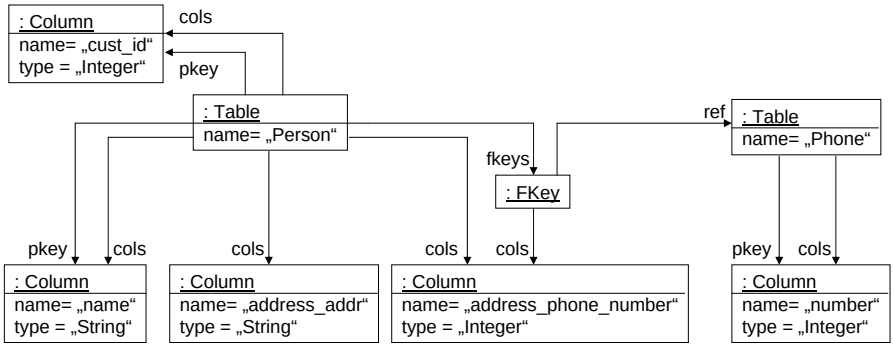

Fig. 2. Data base model as meta-model instance

3 A Model Transformation Approach using Algebraic Graph Transformation in AGG

3.1 The approach

Describing a model transformation by graph transformation, the source and target models have to be given as graphs. Performing model transformation by graph transformation means to take the abstract syntax graph of a model, and to transform it according to certain transformation rules. The result is the abstract syntax graph of the target model.

A model transformation can be precisely defined by a graph transformation system $GTS = (T, R)$ consisting of a type graph T and a set of transformation rules R. The abstract syntax graphs of the source models can be specified by that subset of instance graphs over a type graph T_S which are generated by the corresponding syntax graph grammar. Correspondingly, the abstract syntax graphs of the target models are specified by a subset of instance graphs over a type graph T_T. Both type graphs T_S and T_T have to be subgraphs of the common type graph T. See Figure 3. Starting the model transformation with instance graph G_S typed over T_S, it is also typed over T. During the model transformation process the intermediate graphs are typed over T. Please note that this type graph may contain not only T_S and T_T, but also additional types and relations which are needed for the transformation process only. The result graph G_T is automatically typed over T. If it is also typed over T_T, it fulfills one main requirement to be syntactically correct. If both languages are defined by

syntax graph grammars and assuming that the model transformation starts with a syntactically correct source graph, the result graph of the model transformation should be a syntax graph of the target language, i.e. should be creatable by the target language grammar. In this case, it fulfills all requirements for syntactic correctness.

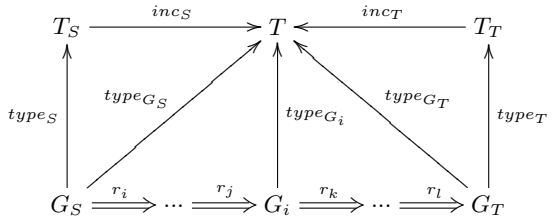

Fig. 3. Typing in the model transformation process

As in object-oriented modelling, types can be structured by an inheritance relation [3]. Instances of a type graph are object graphs equipped with a structure-preserving mapping to the type graph. A class diagram can thus be represented by a type graph plus a set of constraints over this type graph expressing multiplicities and maybe further constraints.

In this approach, models are described by typed attributed graphs. An attribute is declared just like a variable in a conventional programming language: we specify a *name* and a certain *type* for the attribute, and then we may assign any *value* of the specified type to it. All graph objects of the same type also share their attribute declarations, i.e. the list of attribute types and names; only the values of the attributes may be chosen individually. From a conceptual point of view, attribute declarations have to be considered as an integral part of the definition of a type. In theory ([11]), the attribute values are defined by separate data nodes which are elements of some algebra. In the AGG [1] tool, the attribution is based on Java.

A *graph transformation rule* $r : L \to R$ consists of a pair of T-typed graphs L, R such that the union $L \cup R$ is defined. In this case, $L \cup R$ forms a graph again, i.e. the union is compatible with source, target and type settings. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. $L \cap R$ defines a graph part which has to exist to apply the rule, but which is not changed. $L \setminus (L \cap R)$ defines the part which shall be deleted, and $R \setminus (L \cap R)$ defines the part to be created. To make sure that newly created items are not already in the graph, we have to generate new vertex and edge identifiers whenever a rule is applied. Formally, for each application a new rule instance is created. Furthermore, a rule may specify attribute computations. For this purpose, the rule graphs can be attributed by elements of term algebras which are instantiated by concrete values in the graphs when the rule is applied.

A *graph transformation step* is defined by first finding a match m of the left-hand side L in the current object graph G such that m is structure-preserving and type compatible. If a vertex embedded into the context, shall be deleted, dangling edges can occur. These are edges which would not have a source or target vertex after rule application. There are mainly two ways to handle this problem: Either the rule is not applied at match m, or it is applied and all dangling edges are also deleted.

The applicability of a rule can be further restricted, if additional application conditions have to be satisfied. A special kind of application conditions are *negative application conditions* which are pre-conditions prohibiting certain graph parts. This and more general kinds of application conditions are presented in [10].

Performing a graph transformation step with rule r at match m, all the vertices and edges which are matched by $L \setminus (L \cap R)$ are removed from G. The removed part is not a graph in general, but the remaining structure $D := G \setminus m(L \setminus (L \cap R))$ still has to be a legal graph, i.e., no edges should be left dangling. This means if dangling edges occur during a rule application, they have to be deleted in addition. In the second step of a graph transformation, graph D is glued with $R \setminus (L \cap R)$ to obtain the derived graph H. Since L and R can overlap in a common graph, its match occurs in the original graph G and is not deleted in the first step, i.e. it also occurs in the intermediate graph D. For gluing newly created vertices and edges into D, graph $L \cap R$ is used. It defines the gluing items at which R is inserted into D. A *graph transformation*, more precisely a graph transformation sequence, consists of zero or more graph transformation steps.

Given a host graph and a set of graph rules, two kinds of non-determinism can occur: First several rules might be applicable and one of them is chosen arbitrarily. Second, given a certain rule several matches might be possible and one of them has to be chosen. There are techniques to restrict both kinds of choices. Some kind of control flow on rules can be defined by applying them in a certain order or using explicit control constructs, priorities, etc. Moreover, the choice of matches can be restricted by specifying partial matches using input parameters. A common form of controlled rule application is the following one: One rule is selected from outside (e.g. the user) and triggers the application of a number of other rules which become applicable after the first rule has been applied.

3.2 The solution

In the following, we specify a model transformation from class diagrams to relational data base models by algebraic graph transformation. The solution is presented in two steps: First a basic solution is presented not considering class inheritance and non-persistent classes. Thereafter, we discuss a complete solution which satisfies all requirements stated in Section 2.

A basic solution First of all, we present the type graph for the chosen model transformation in Figure 4. It consists of a type graph for class diagrams (the left part) which follows the meta-model for class models given in the workshop announcement [4] where the inheritance is flatten. In the advanced solution it will follow the class meta-model directly. Furthermore, it contains a type graph for the relational data base model (the right part) which also correspond exactly to the meta model for relational data base models given in the announcement. For the model transformation, some additional helper structure (the part in between) is needed which mainly holds correspondences between elements of different type graphs.

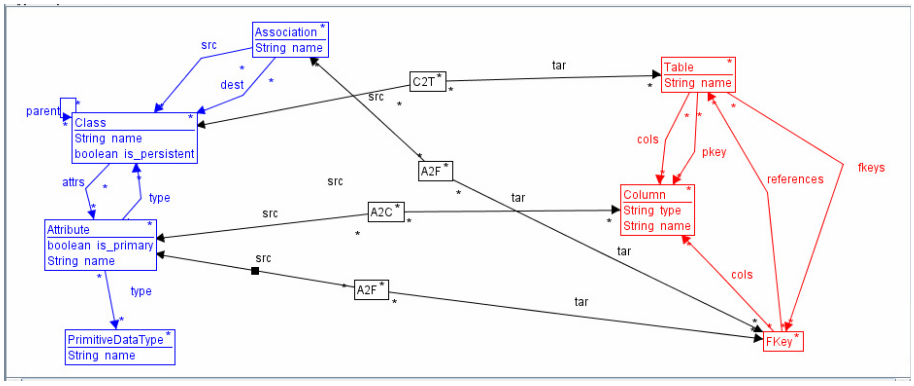

Fig. 4. Basic type graph for the sample model transformation

We start the model transformation by mapping persistent classes to tables. For each class a table with the same name is created and the correspondence between both is set. This step is formulated by rule "Class2Table" in Figure 5. This figure shows the left and the right-hand sides of the rule. The mapping between both rule sides is indicated by the same number at both classes. This rule states that the class is preserved, while the table and the correspondence structure are created. This rule and all the following ones in this subsection have a negative application condition (NAC) which is equal to the right-hand side. NACs of this special form are used to apply a rule only once at the same match.

Next we consider the transformation of class attributes performed by the application of rule "PrimAtt2Col" in Figure 6. Given a transformed class, an attribute of this class is mapped to a column of the corresponding table. The column is named as the attribute. Here, we consider the case where the attribute type is a primitive data type. In rule "AttPersClass" in Figure 7 we consider also the transformation of attributes, but this time the attribute type is a persistent class. In this case, a corresponding table exists also for this class, together with a primary key. Rule "AttPersClass" inserts a foreign key to this table and a new

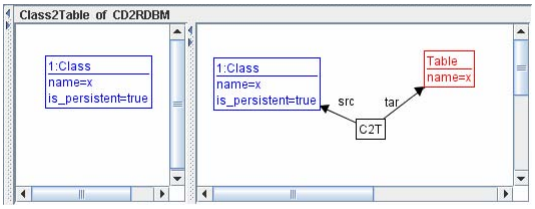

Fig. 5. Rule "Class2Table"

column. The name of this new column is a concatenation of the attribute name and the primary key of the foreign table.

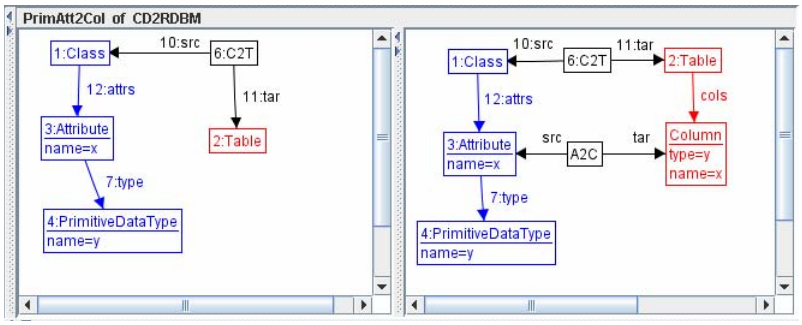

Fig. 6. Rule "PrimAtt2Col"

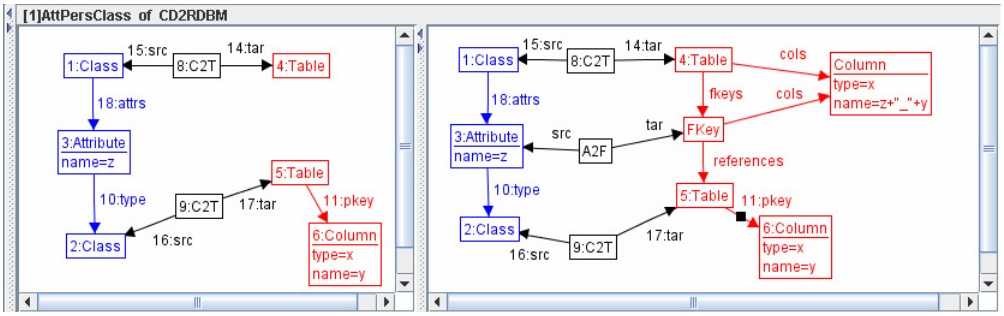

Fig. 7. Rule "AttPersClass"

Rule "setkey" in Figure 8 sets a primary key to that column which is the translation of a primary attribute.

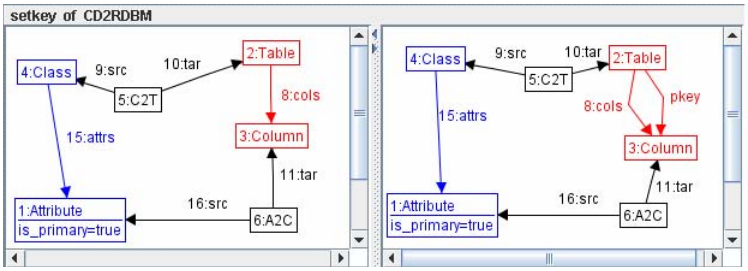

Fig. 8. Rule "setkey"

Finally, rule "AssPersClass" in Figure 9 translates an association between two classes in the same way as an attribute with a class as type was translated. That means a foreign key to the corresponding table of the destination class is created together with a new column. The name of the column is the association name followed by the primary key of the foreign table.

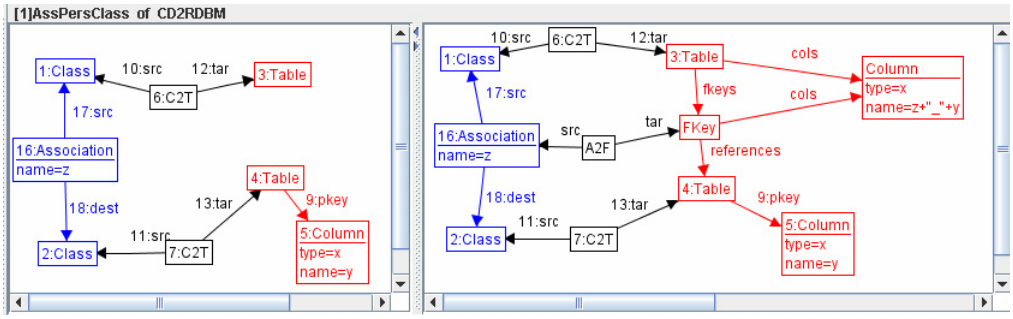

Fig. 9. Rule "AssPersClass"

Rules which delete the helper structures as well as the source model are optional. Since it should be clear how they would look like, we omit them here.

An advanced solution The complete model transformation builds up on the basic solution, but takes also class inheritance and non-persistent classes into account. In class hierarchies, only the top-most classes are mapped to tables, while additional information given by subclasses has to be merged in. That means additional attributes and associations of subclasses are mapped to additional columns in the corresponding table.

To handle this requirement, we first compute the transitive closure of the inheritance relation (expressed by "parent" edges) and store it as additional helper structure in the instance graph (using "anc(estor)" edges). Furthermore, we pull up attributes and associations from the subclasses to the top-most classes. To

distinguish these additional attributes and associations we add two helper types: "Att" and "Assoc" (with additional edge types). Moreover, we use node type inheritance [3] and multiplicities here, as used for class models in the given meta-model in [4]. The extended type graph is shown in Figure 10.

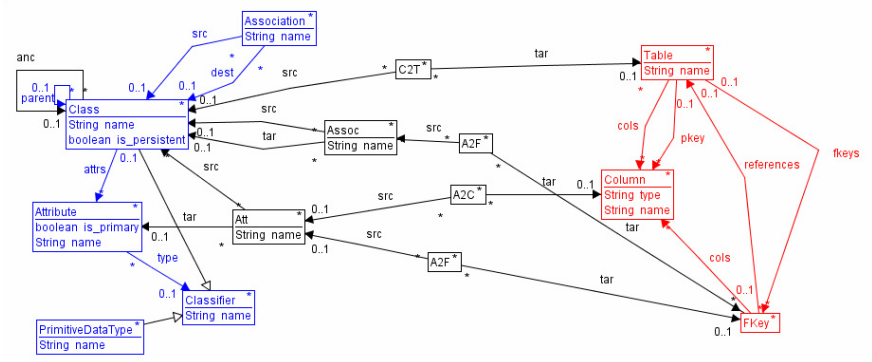

Fig. 10. Type graph for the sample model transformation

In Figures 11 and 12 the corresponding rules are shown which first add an anc(estor) edge for each parent edge and then compute the transitive closure in form of additional helper edges by recursive rule application. By recursively applying the rule in Fig. 13 all subclass' attributes are collected in the topmost classes. After having transformed the topmost classes to tables, their subclasses are also transformed to their tables by the recursive application of the rule in Figure 14. Please note that Fig. 11 - 15 show rules with three compartments: The leftmost compartment shows a NAC, in the middle the LHS is depicted and on the right we see the RHS.

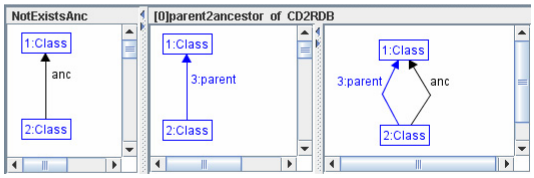

Fig. 11. Rule which initializes the transitive closure computation for class inheritance

Non-persistent classes are not mapped to tables, however, one of the main requirements for the model transformation considered is the preservation of all the information of the class diagram. That means having attributes or associations pointing to non-persistent classes, all attributes and associations of these classes

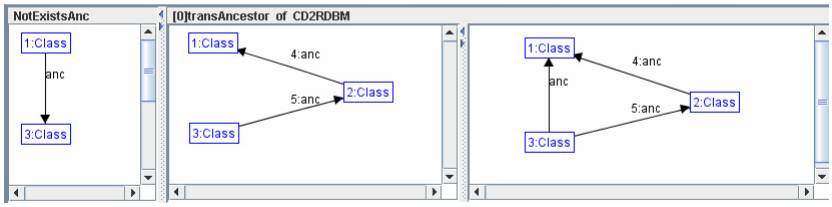

Fig. 12. Rule which computes the transitive closure for class inheritance

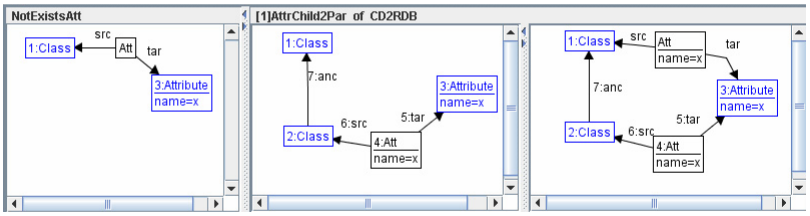

Fig. 13. Rule which collects attributes of subclasses in their parent classes

are also mapped to columns belonging to the table of the original persistent class.

Since the non-persistent structure can be arbitrarily large, we first collect all attributes and associations of non-persistent classes in those persistent classes pointing to the non-persistent ones. These additional attributes and associations are also stored in the helper types "Att" and "Assoc". The name of an additional attribute (association) is the concatenation of names occurring on the path from the persistent class to this attribute (association), including its own name.

The rule in Figure 15 collects all attributes of non-persistent classes which in their referencing classes. By recursive rule application, the attributes are stepwise added to the referencing classes until they are persistent (see NAC "NoAncestor"). This rule has another NAC equal to the RHS (which is not shown). Associations can be handled in the same way. The corresponding rules are not shown within this paper.

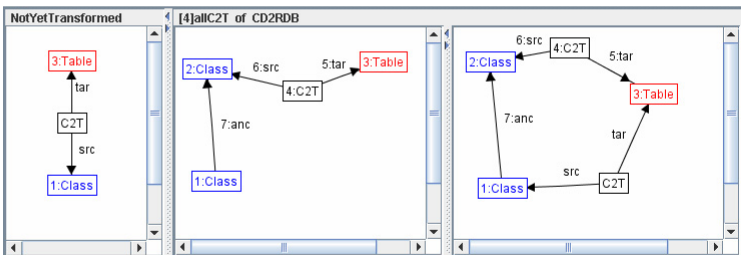

Fig. 14. Rule which transforms a subclass to its ancestor's table

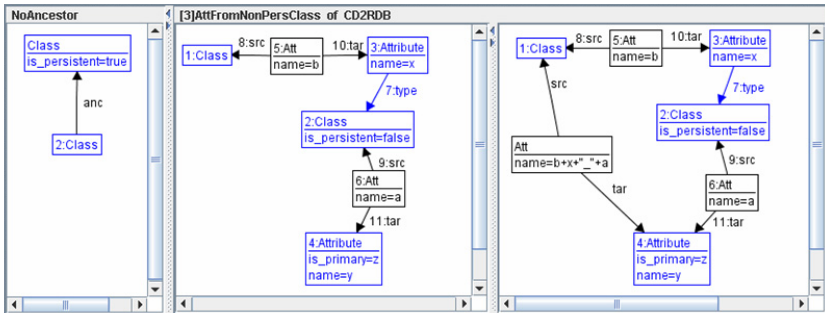

Fig. 15. Rule which collects attributes of non-persistent classes in those classes referring to them

In the complete model transformation we first build up the necessary helper structure by additional rules. Then, the actual model transformation takes place, using the rules presented above in principle. They have to be slightly adapted to handle also those additional attributes and associations coming from subclasses or connected non-persistent classes.

In this solution, we do not define additional control structures for the rule execution, but coordinate them by the definition of layers only. Each rule is assigned to a certain layer. Starting with layer 0, the rules of one layer are applied as long as possible. Thereafter, the next layer is executed. Having executed the highest layer, the transformation is finished. In the complete model transformation we distinguish the following layers:

0 - computation of transitive closure of class inheritance
1 - collection of attributes and associations of subclasses
2 - collection of attributes and associations of connected non-persistent classes
3 - creation of tables
4 - creation of columns
5 - deletion of helper structure
6 - deletion of source model

For this model transformation example the use of layers was enough to control the rule application. More complex model transformations might use transformation units which have been introduced by Kreowski and Kuske in [14]. Transformation units provide the means to structure a system of rules to subcomponents that can use each other. Within a unit, the set of rules of another unit can be 'called' by using the name of the corresponding unit, thus realizing functional abstraction.

3.3 Validation of Model Transformations

Due to its formal foundation [11], the algebraic graph transformation approach allows to validate model transformation concerning different properties.

Following the model transformation approach described above (see Figure 3), the result graph G_T is automatically typed over graph T. If it is also typed over T_T, it fulfills one main requirement to be syntactically correct. Otherwise, additional model elements occur which could be deleted by special deletion rules.

If both languages are defined by syntax graph grammars and assuming that the model transformation starts at a syntactically correct source graph, the result graph of the model transformation should be a syntax graph of the target language, i.e. should be creatable by the target language grammar. In this case, it fulfills all requirements for syntactic correctness. If the target language is just defined by a set of constraints over a given type graph, the result graph has to fulfill these constraints. In the case of graph constraints, there is a theory ([10]) on transforming constraints to corresponding application conditions for rules in the following sense: Starting with a correct source model, all rules (possibly extended by additional application conditions) can only be applied such that the result graphs are consistent with respect to the given graph constraints.

Besides syntactical correctness, model transformations should show further correctness issues. A model transformation should lead to a unique result, i.e. it should show a functional behaviour. This means not only that the transformation rules should not be applied in a way that several results are possible, i.e. the transformation system must be confluent , but also that all model transformations have to terminate. Corresponding criteria are given in [11] for confluence and [7, 6] for termination.

3.4 The Tool Support

The approach presented is supported by AGG [1] which is an integrated development tool for typed attributed graph transformation, implemented in Java. It offers the visual development of graph transformation systems including visual editing and simulation as well as a number of validation tools. The internal graph transformation engine can also be used by a Java API and thus, can be integrated into other tool environments.

Since the theoretical concepts are implemented as directly as possible – naturally, respecting necessary efficiency considerations – AGG offers clear concepts and a sound behavior concerning the graph transformation part. Since graph transformation can be applied on very different levels of abstraction, it can be non-attributed, attributed by simple computations or by complex processes, depending on the abstraction level. To reflect this wide application area for attributed graph transformation, we decided to attribute AGG graphs by Java objects. On one hand, this design decision certainly allows a large variety of graph transformation applications, but on the other hand it is clear that the Java semantics is not covered by the formal foundation.

Rule applications can be performed in two different modes: the interactive mode and interpreter mode. In the interactive mode, the rule selection and match definition can be done interactively. After having chosen a rule, the match can either be given element-wise by the user, or computed automatically. If several matches are possible, they are computed one after the other. A third possibility

is the manual definition of a partial match which is completed automatically afterwards. The automatic match completion computes the possible matches in an arbitrary order which certainly differs from the order when repeating the computation of all possible matches. After having fixed rule and match, the rule is applied to the current host graph. The second possible mode for rule application is the interpreter mode where rules are applied as long as possible. The rule as well as the match selection orders are non-deterministic, except for layered graph grammars or other kind of rule control as by Java programs using the AGG API.

Due to its formal foundation, AGG offers validation support including consistency checking of graphs and graph transformation systems, critical pair analysis and termination checks for graph transformation systems.

4 A Triple Graph Grammars Approach with AToM^3

4.1 The approach

Triple Graph Grammars [21] were proposed by Andy Schürr as a means to specify translators of data structures, check consistency, or propagate small changes of one data structure as incremental updates into another one. Triple graph grammar rules model the transformations of three separate graphs: source, target and correspondence. Nodes in the latter have a couple of morphisms, one to nodes in the source graph and another one to nodes in the target graph. In our approach, we allow the morphisms to be partial. In this way, the rewriting is performed on triple graphs, which have the form $SOURCE \xleftarrow{s} CORR \xrightarrow{t} TARGET$. In a similar way as regular graph rules (see section 3), triple graph grammars are made of triple rules, each having triple graphs in its left and right hand sides (LHS and RHS). When a matching (three morphisms in our case) is found between the rule's LHS and the triple graph, then the occurrence can be substituted by the RHS.

Triple graph grammars can be extended by providing a triple meta-model for typing the triple graphs [12]. This triple meta-model describes the kind of structure that can be found in the source, target and correspondence models. In our case, we consider meta-models containing inheritance relations and additional textual constraints.

In our approach we also provide rules with application conditions [10], restricting the context in which they can be applied. These application conditions were developed for regular graph transformation, but we have adapted them for triple graph grammars. A condition has a premise graph X, a set of consequent graphs Y_i, and morphisms y_i from X to each Y_i: $c = \{X, X \xrightarrow{y_i} Y_i\}$. If an occurrence is found for X in the host graph, then an occurrence has to be found for some consequent graph Y_i for the rule to be applied. If a condition does not have consequent graphs, finding a match for the premise forbids the rule application. This is a special case of condition called *negative application condition* (NAC). On the contrary, if the condition has an empty premise, it is a positive application condition.

For rule execution, we use the control structure provided by AToM^3 [15]. This control structure is based on rule priorities. Rules with the same priority are chosen at random. The rewriting engine tries first the rules with the highest priority. When a rule with a certain priority is executed, the control goes back to the rules with the highest priority. If no rule with a certain priority can be executed, then the control goes to the next lower priority. The grammar execution ends when no rule (with the lowest priority) can be executed. In addition, it is possible to select a list of graph transformation systems to be executed in sequence.

4.2 The solution

The solution we present here is based on the AToM^3 tool [15], which supports regular graph grammars (in the style of the ones presented in section 3) and triple graph grammars. For the example, we use triple graph grammars and discuss its merits with respect to the AGG solution based on regular graphs.

The source and target meta-models are the ones in the specification. In the correspondence meta-model we do not explicitly create a meta-model, but use untyped graphs. This is a feature of AToM^3 (see next subsection). We allow correspondence nodes to relate columns and attributes, classes and tables, two columns (by connecting two correspondence nodes and each one of them to the columns, see Figure 18), and two foreign keys. Note how, the partial morphisms from the correspondence nodes allow relating elements from one model only.

The main idea of our solution is given in the following list of steps, where each step is made of one or more rules. Rules in steps 1-8 have the same priority, so no order should be assumed. Nonetheless, due to the NACs in the different rules, the execution order is similar to the ordering we present in the list. We show pictures of some representative rules only.

1. Create a table for each top-most class, even if the class is non-persistent, and relate such table with the class (one rule). The relation is done through a node in the correspondence graph. This rule is similar to rule "class2table" shown in Figure 5 in the AGG section.
2. For each attribute of a class already related to a table, create a column in the corresponding table (one rule). The rule is not executed if there is already a column with the same name in the same table. Figure 16 shows this rule. In each graph, the upper part is the class diagram model, while the lower part is the relational model. The rule has two NACs, the first one checks that the attribute has not already an associated column. The second one is useful as later (in step 3), subclasses may be related with the table, and new columns may be generated. We do not generate a new column for an overriding attribute, but keep the generated column by the attribute in the parent class. In this way the NAC forbids the rule execution if the table has already a column with the same name as the attribute. Note how this rule is similar to rule "PrimAtt2Col" in Figure 6 in the AGG section. Note however that in the triple rule, the elements of the different models are arranged in the different graphs.

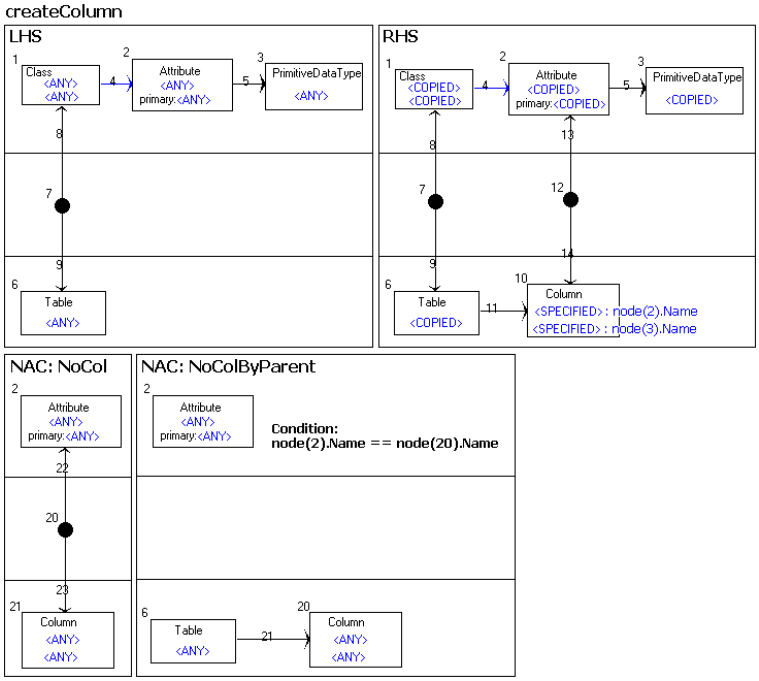

Fig. 16. createColumn rule (step 2).

3. For each subclass, relate the class with the table of the top-most parent (one rule). Note how the execution of this rule may trigger the execution of the rule in step 2. The rule is shown in Figure 17. The rule is iterativelly applied down the inheritance hierarchy (association labelled as 5 in the rule). The rule has a NAC that allows executing the rule only once for each class (it checks if the class already has an associated table). In addition, the rule has an application condition that forbids its application if a column has not been created for some attribute in the parent class. The condition can be read as: "if the class has an attribute (graph X), then it should have an associated column (graph Y)". In this way, this rule is not applicable if the rule in step 2 can still be applied at some match.
4. Create the primary keys (one rule). This rule is similar to the "setkey" rule in the AGG approach (see Figure 8).
5. For each related persistent class (through an association or attribute), create the corresponding foreign key (four rules, two for attributes and two for associations). The rules are similar to the "AssPersClass" of the AGG approach (see Figure 9).
6. For each related non-persistent class, copy the columns (two rules, one for attributes and one for associations). One of the rules in this step (the one for attributes) is shown in Figure 18. The copied column is related to the original one by means of two nodes in the correspondence graph. The first NAC

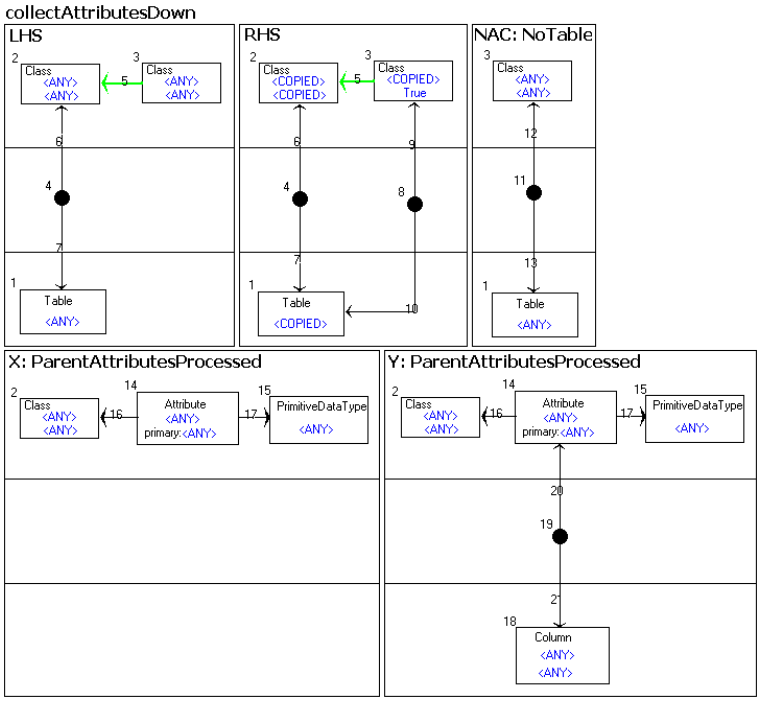

Fig. 17. collectAttributesDown rule (step 3).

forbids the rule execution if the column has already been copied (the columns are already related). The second NAC forbids the rule execution if the table already has a column with the appropriate name. This NAC is necessary to allow incremental execution of the grammar. It should be noticed that the tables for non-persistent classes (and their related columns) will be deleted after the grammar execution. In this way, if the user adds elements in the class diagram and then executes again this grammar to update the relational model, the tables and columns for non persistent classes would be created again. Without this NAC, the column would be copied again in the persistent class.

7. Add the primary keys for the newly added columns by the rules in step 6 (one rule). Note how the execution of this rule may trigger the execution of rule number 5, to update the foreign keys. Figure 19 shows the rule for this step. The rule looks for two columns (where exactly one of them is a primary key –labelled "7" in the picture–) related by nodes in the correspondence graph (which means that one of the columns was copied). Then, a primary key arrow (the red one, labelled "13") is added to the column that is not a primary key.
8. Copy the foreign keys from the newly added columns by the rules in step 6 (two rules).

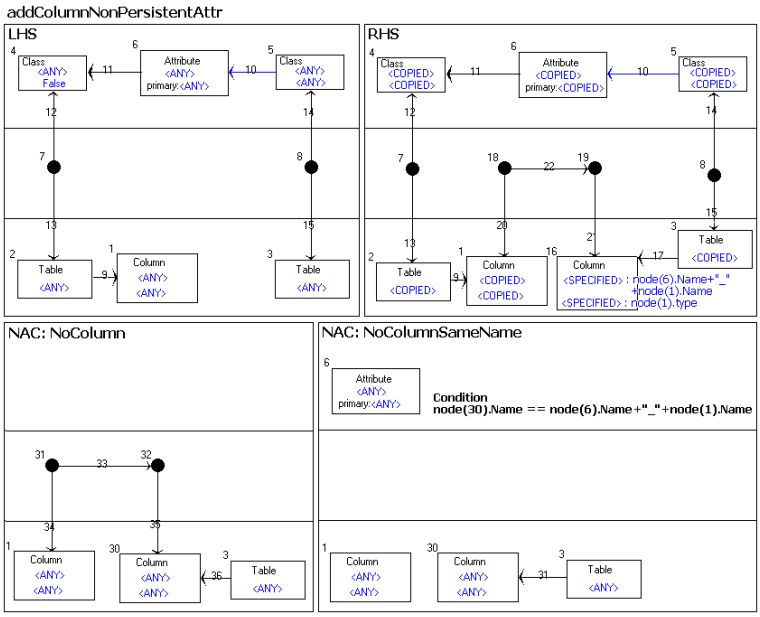

Fig. 18. AddColumnNonPersistentAttr rule (step 6).

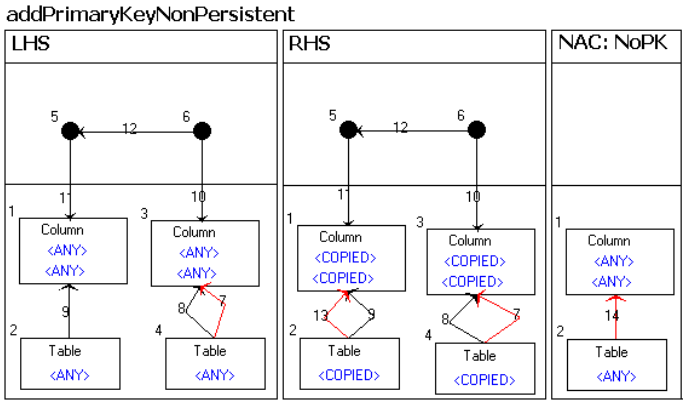

Fig. 19. AddPrimaryKeyNonPersistent rule (step 7).

9. Delete the tables and columns associated to non-persistent classes (twelve rules). These rules should be executed once no other rule is applicable. Therefore they have been implemented in a separate transformation that is executed once the previous one ends. The deletion of elements needs so many rules because we are using the so-called Double Pushout (DPO) approach [9]. This approach forbids deleting a node if it has some incoming or outgoing edge. Therefore, before deleting a node, we have to execute rules to delete

all surrounding edges. On the other hand, the Single Pushout (SPO) approach [9] allows the deletion of nodes with incident edges, and the edges are deleted as well. Although both approaches are supported by AToM^3, we usually prefer the more conservative DPO as in this case a rule execution does not have side effects.

One of the differences with the AGG solution is that we implement the transitive closure of inherited attributes by directly creating the columns in the associated table, and then performing the rest of the operations with the columns. Moreover, we temporarily create tables for non-persistent classes. These differences are not essential however, and the AGG solution could have been implemented with triple graph grammars as well. Other deeper conceptual differerences are discussed in the next subsection and in section 7.

4.3 The Tool Support

In this section we show the implementation of the proposed solution in the meta-modelling and transformation tool AToM^3 [15]. This tool allows the specification of Domain Specific Visual Languages by means of meta-modelling, and their manipulation by means of graph transformation. Recently, AToM^3 has been provided with the possibility to define triple graph grammars [12] and multiple views [13]. Views provide their own meta-model, which optionally can be subsets of a global meta-model that relates all the view concepts. This is very useful when defining multi-view languages (especially if such views contain overlappings), such as UML. We have used this AToM^3 feature to define two independent views: one for the Class Diagram and one for the Relational Database model. We defined the corresponding sample meta-models in the views.

In addition, AToM^3 allows the definition of consistency relations between views. These consistency relations contain a triple graph grammar that can be executed when the view source of the relation changes. We have defined two relations that have the Class Diagram view as their source. The first one contains a triple graph grammar for generating and updating the relational model (containing rules for steps 1-8). The second one contains another triple grammar that deletes the tables for non-persistent classes (rules for the last step). It is executed after the previous one has finished. A screenshot of the definition of the environment is shown in Figure 20.

The picture shows in the bottom window the two views and the consistency relations. One of the relations (the one that does the main computation) is being edited in the rest of the windows. The dialog window on top of the previous one allows editing the relation name (called *ClassDiagram2Relational*). The next window allows editing the triple graph grammar. The LHS of one of the rules (*MergeFK1*, step 8) is being edited. The rule copies the foreign key of a table related to a non-persistent class. All attributes in the LHS are set to "any" meaning that any value in the model will make a match. In the RHS, attribute values can be copied from the LHS or specified by means of Python code.

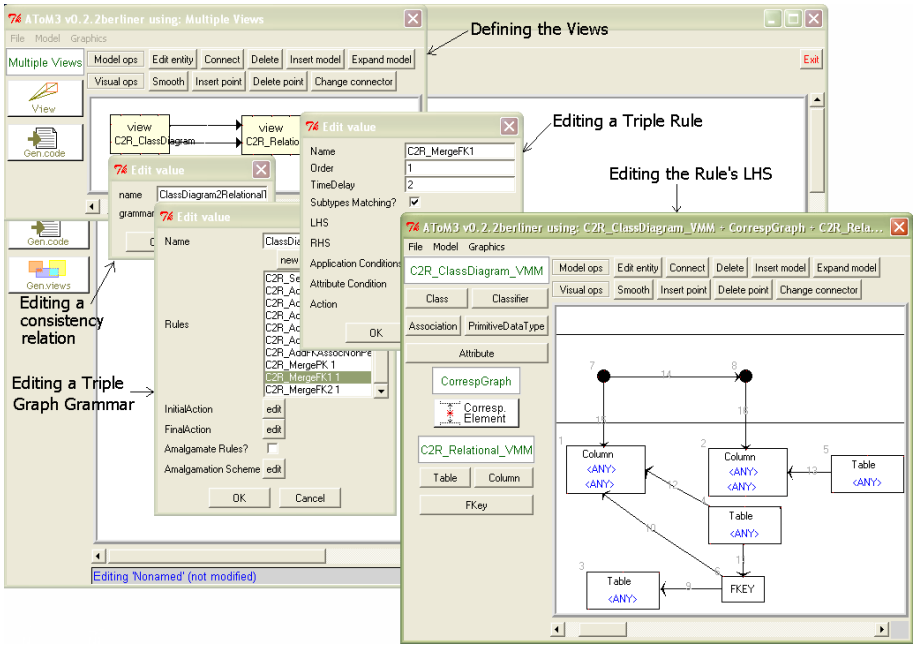

Fig. 20. Defining the class to relational environment with AToM^3.

From this specification, AToM^3 generates a tool where the user can insert class diagrams and relational models. Nonetheless, for this application, we are interested in generating the relational model from the class diagram models. After editing a class diagram, the user may run the consistency relations to create or update the associated relational model. A screenshot of this environment is shown in Figure 21. The window at the bottom allows inserting relational and class diagram models. A class diagram is being shown in the window to the left. It has been converted to a relational model, which is shown in the window to the right. The correspondence graph between the class diagram and the relational model is hidden to the user. This is because it is only used internally to maintain the consistency between both models.

One of the advantages of the approach in this section is the modularity, as it keeps separated the source and target models, which are related through a separate model. That is, the class diagram and the relational database are two completely separated models. Moreover, the meta-models remain also separated and do not have to be modified. In this way, it could be possible to define an environment with different model transformations from/to a certain domain. The triple graph grammars approach is cleaner than mixing all the meta-models and models into a single one. This has another advantage: in the regular graph transformation approach an important aspect is to verify whether the target model is correctly typed over the target meta-model. That is, at the end of the transformation no source or auxiliary elements should remain. This is no longer a

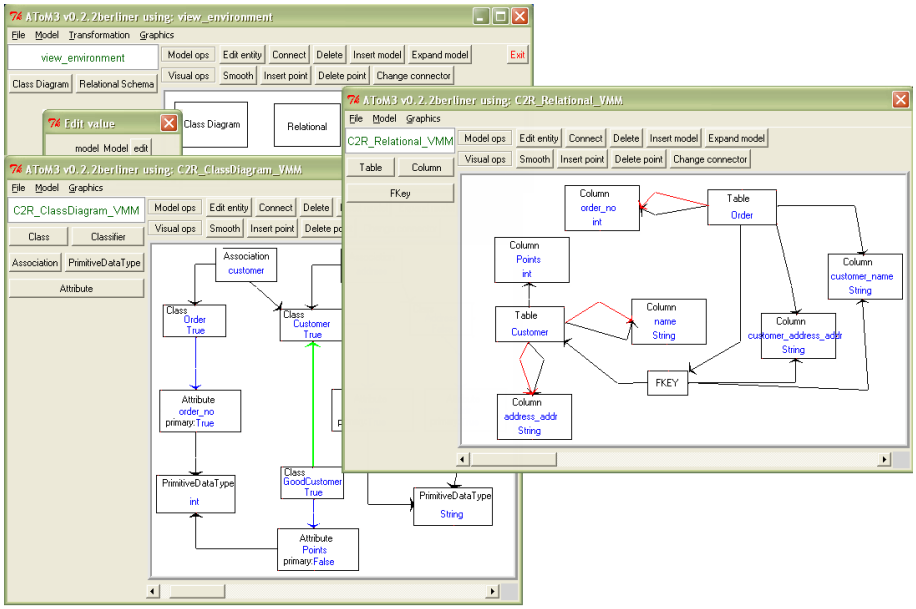

Fig. 21. The class to relational environment generated by AToM^3.

problem in the triple graph grammars approach, as the target model is correctly typed because the rules are correctly typed and no auxiliary elements are present in the target model (they are kept in the correspondence graph).

Another main difference with the regular graph transformation approach is that we do not delete the source elements and keep the correspondence graph in between the source and target. This allows for incremental transformations (see discussion in section 7).

5 Controlled Graph Transformation with Recursive Graph Patterns in VIATRA

In many graph transformation approaches (e.g. Progres, Fujaba, Great, VIATRA or VMTS), the application of elementary algebraic graph transformation rules is frequently driven by complex control structures to reduce non-determinism and thus to improve run-time performance. In addition, graph patterns allowed in the left-hand side of rules are also more powerful than in the basic algebraic approach by allowing so-called path expressions or multi-objects. The expressiveness of similar extensions is demonstrated in the paper on two approaches, namely, VIATRA (Sec. 5) and VMTS (Sec. 6).

5.1 The Approach

The VIATRA approach combines the rule and pattern-based paradigm of graph transformation (GT) and the very general, high-level formal paradigm of *abstract state machines (ASM)* [5] into a single framework for capturing transformations within and between modelling languages (following the results of [23]). On the semantic level, this combined VIATRA specification is translated back into ASMs, which thus serve as a very high-level, executable virtual machine.

Model definition. VIATRA uses the *VPM metamodeling* approach [24] for describing modeling languages and models, which supports *arbitrary meta-levels in the model space*. As a direct consequence, models taken from conceptually different domains (and/or technological spaces) can be easily integrated into the VPM model space. In the VIATRA section of the paper, models and modelling languages will be denoted in the traditional MOF notation for presentation reasons.

Queries on models. Queries on models are intuitively captured by *generalized (recursive) graph patterns*. Graph patterns in VIATRA may contain an arbitrary number of negative application conditions marked with the neg keyword, which denotes that a graph pattern becomes a negative (forbidden) pattern. It is a generalization of traditional negative application conditions in the algebraic GT approach (see Sec. 3) in a sense that negative patterns may in turn contain negative patterns (negation with arbitrary depth). Furthermore, patterns are allowed to call call other patterns even in a recursive way (using the find keyword).

Elementary model manipulations. Elementary model manipulations are specified by graph transformation rules. A graph transformation rule in VIATRA consists of a *precondition pattern* and a *postcondition pattern* both of which are arbitrary graph patterns basically corresponding to the LHS and RHS of the algebraic GT rules. To support the modular design of transformations and to increase reusability, pre- and postcondition of GT rules may refer to predefined (named) graph patterns. (See the definition of rule subclassR in Fig. 22.)

Furthermore, input (and output) parameters may be passed to a VIATRA rule or a pattern to improve performance and to define the morphism between pre- and postcondition graph elements. If a graph element is passed to the postcondition pattern as a parameter, but it does not appear there, it prescribes the deletion of the matched element. Consequently, if a graph element appears only in the postcondition (but not in the precondition), a new model element is created.

As a consequence, the main conceptual difference on the rule level compared to the basic algebraic approach is that only those parts of the precondition pattern need to be passed as parameters to the postcondition pattern, which are (i) either nodes in the interface graph (formally, in $L \cap R$) or (ii) they are nodes and edges to be removed (thus, in $L \setminus R$). Naturally, it is also possible to spuriously pass the entire precondition to the postcondition using different parameters.

A unique feature of VIATRA is the support of *generic and meta-transformations* [25] that allow type parameters and manipulate transformations as ordinary models, respectively. This allows to arrange common graph algorithms (e.g. transitive closure, graph traversals, etc.) into a reusable library, which is called by assigning concrete types to type parameters in the generic rules.

Complex transformation programs. Complex transformation programs are assembled by using abstract state machine constructs that provide higher-level control structures for elementary manipulation steps.

ASMs provide complex model transformations with many powerful control structures including the sequencing operator (seq), rule calls to other rules (call), variable declarations and updates (let and update constructs) and if-then-else structures, non-deterministic selected (random) and executed rules (choose), iterative execution (applying a rule as long as possible), and the deterministic parallel rule application at all possible matchings (locations) satisfying a condition (forall). In the sequel, we use the collective term "*ASM rules*" to denote one of these constructs responsible for the control flow of the transformation. Examples on ASM rules will be given in Fig. 23 later on.

Both ASM and GT rules are allowed to have input and output parameters to support information hiding and to improve the efficiency of pattern matching.

The semantics of an ASM rule is defined by a set of consistent elementary update steps manipulating the model space (or other runtime ASM data structures).

Interaction between ASMs and GT rules. Abstract state machine rules (defined by the rule keyword) and graph transformation rules (defined by the gtrule keyword) interact with each other in a mutual way.

- *ASM rules using GT rules.* ASM rules may initiate the application of graph transformation rules by quantifying input parameters of GT rules existentially or universally. In the latter case (extensively used in the VIATRA part of the paper), all tuples of the quantified variables satisfying a guard (which is a Boolean ASM expression or a graph pattern) are enumerated and the GT rule is applied to all tuples in a single parallel step.
  Finally, when a GT rule is applied successfully, the output parameters of the rule get instantiated (by the RHS). These are returned to the ASM rules by parameter passing along variables.
- *GT rules using ASM rules.* A graph pattern in a GT rule may contain a check part, which allows to check attribute conditions. Furthermore, a GT rule may also contain an action part, which is a sequence of ASM rules executed after the model manipulation part of the GT rule has completed for a given matching.

Code generation. Code generation is treated as ordinary model-to-code model transformations, and it is supported by intelligent (model-driven) print ASM

rules, and a code formatter mechanism to split the generated code into different source files.

Further conceptual details of the VIATRA approach are discussed together with the solution in the upcoming section.

5.2 The Solution

As the main guidelines of the solution was already presented both using the basic algebraic approach (Sec. 3) and triple graph grammars (Sec. 4), in the VIATRA solution, we focus only on three critical parts of the transformation. More specifically, we discuss

1. the *transitive closure of ancestor edges* starting from basic parent edges;
2. the *merging-in of associations* in a chain of non-persistent classes;
3. the *generation of columns and foreign keys* from such associations (and attributes).

We believe that these steps are interesting and representative as they solve the selected subproblems of the entire transformation in a more succinct or efficient way. The remaining parts of the model transformation in VIATRA conceptually follow the basic algebraic case (naturally, seq, iterate and forall ASM control structures are used instead of layers).

Graph transformation rules for transitive closure. The transitive closure calculation of ancestor edges requires two steps as described by the two graph transformation rules in Fig. 22.

- First ancestor edges are generated for all parent edges by applying rule parentIsAncR. It is worth pointing out that edge P1 of type parent in the precondition is not passed as a parameter to the postcondition. As a result, the missing parent edge does not mean that P1 is removed by the rule application, since VIATRA only removes model elements which are passed to the postcondition as a parameter but they do not appear in the postcondition.
- Then, according to rule subclassR, if a class C2 is an ancestor of class C1, class C3 is an ancestor of C2, but C3 is not yet an ancestor of C1 (see the negative condition), a new anc edge is generated from C1 to C3. This time, the all elements from the precondition pattern lhs are passed as parameters to the postcondition rhs (but anc edges E1 and E2 could be omitted from the passed parameters as done in rule parentIsAncR).

ASM program for transitive closure. Elementary graph transformation rules are controlled by ASM rules (defined by the rule keyword) into complex transformation programs. The ASM rules of the transitive closure calculation of ancestor edges is presented in Fig. 23.

1. Initially, an anc edge is created for each parent edge (GT rule parentIsAncR applied in *forall* mode), and then ASM rule closureAnc is called.

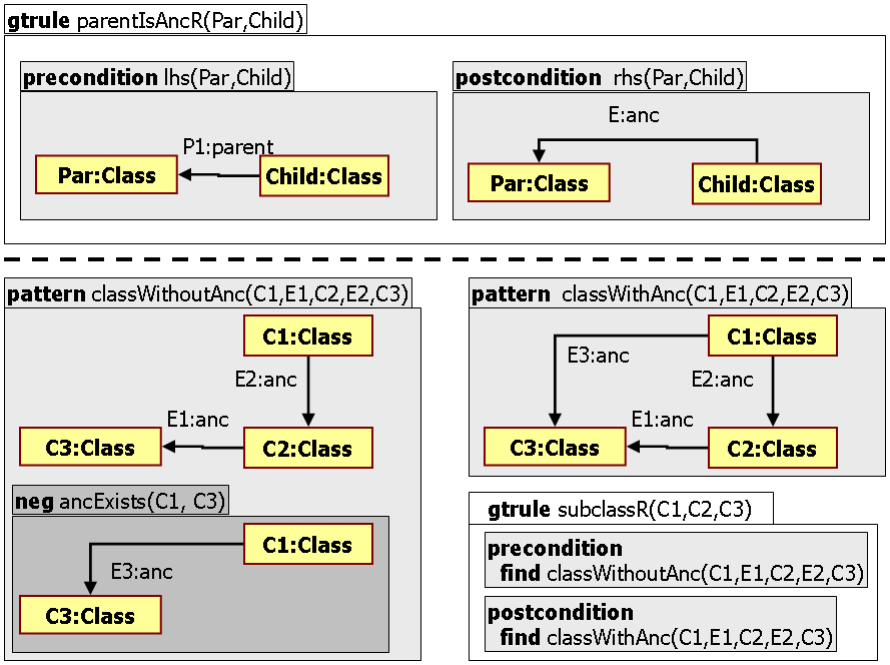

Fig. 22. Graph transformation rules in VIATRA for the transitive closure of ancestor edges

2. In this latter ASM rule, we check if there are triples of classes where only the transitive anc edge is missing (see the conditional construct which calls pattern classWithoutAnc).

```
rule transClosureAnc() = seq {
    forall P,C with parentIsAncR(P,C).pre do
        apply parentIsAncR(P,C);
    call closureAnc();
}
rule closureAnc() =
if (find classWithoutAnc(Cls1,Cls2,Cls3))
    forall C1,C2,C3 with subclassR(C1,C2,C3) do skip;
    call closureAnc();
else skip;
```

```
forall C1,C2,C3 with subclassR(C1,C2,C3).pre do
    apply subclassR(C1,C2,C3);
≡
forall C1,C2,C3 with apply subclassR(C1,C2,C3) do
    skip;
```

Fig. 23. ASM programs for the transitive closure of ancestor edges

3. Then subclassR is applied in parallel to all such classes[5].
4. Finally, rule closureAnc is called recursively.

While in case of the algebraic GT approach, the transitive closure of rule subclassR would be calculated by an iterative rule application (i.e. the rule is applied as long as possible), the forall ASM control structure provides a more efficient solution, which handles all the matches of the rule in parallel.

Obviously, the latter solution is only applicable if there is no overlapping between the different matches of rule subclassR. In the current example, this means that there is no multiple inheritance in the source UML model, which is assumed in the paper, since the database encoding (with a single database column referring to the type of an element) is only applicable in the same case.

Generic transformation rule for transitive closure. Note that the transitive closure of a certain edge type is a frequent subproblem in many model transformations. Therefore, it would be advantageous to use a common reusable library of transformation design patterns as pointed out in [2]. For this purpose, VIATRA offers the concepts of generic model transformation rules as depicted in Fig. 24.

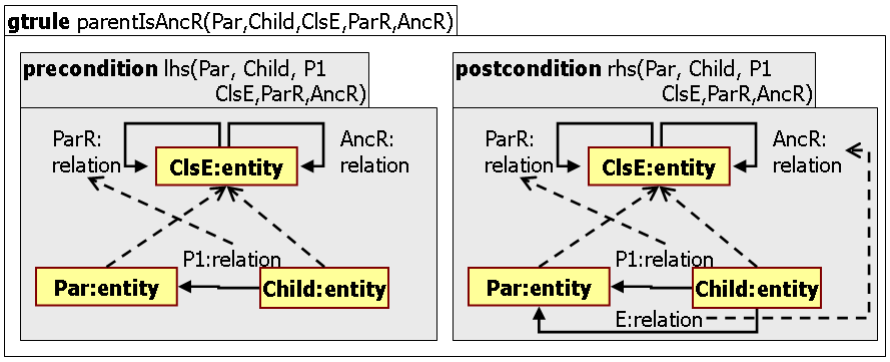

Fig. 24. A generic graph transformation rules in VIATRA for the basic step of transitive closure

Compared to subclassR, this generic rule has three additional input parameters:

- ClsE for the type of the nodes (Class previously),
- ParR for an edge type which denotes the type of the elementary relation to be closed transitively (parent in the previous example), and
- AncR for the transitive edge type (anc in the example).

[5] For efficiency reasons, graph transformation rules are also allowed in the guard of a forall ASM statement, which is equivalent to an atomic step of a pattern matching and a manipulation phase (see the two equivalent solutions for calling subclassR).

Now all pattern elements have types taken from the built-in (top-most level) VPM meta-model, while the type-instance relationship between meta-model and model nodes and edges are denoted by the dashed instance-of arrows.

When interpreting this generic pattern, the VIATRA engine first instantiates the type parameters (i.e. ClsE, ParR, AncR) and then queries all the instances of these types. As a result, the same set of rules can be applied in various modelling languages.

Merging-in of associations. The merging-in of associations is captured by the GT rule assocNPClassAssocR in Fig. 25. Prior to applying rule assocNPClassAssocR, another rule should generate Assoc instances for all Association instances. Then rule assocNPClassAssocR can be applied as long as possible using the iterate construct.

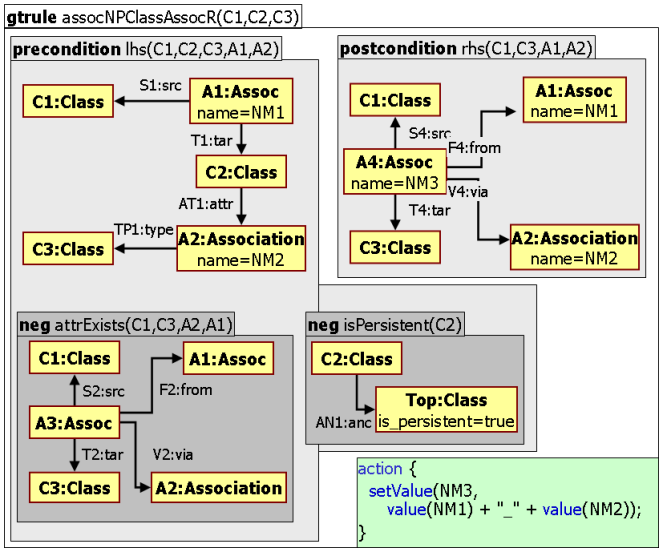

Fig. 25. Merging-in the associations of non-persistent classes

The rule itself finds a two-step path in the association chain (an Assoc object leading to a class C2 with non-persistent ancestors (i.e. its top-most parent class is non-persistent) followed by an Association leading out from C2) which is not yet processed. In the case when different association paths are leading to the same non-persistent class, all associations leading out of this class are merged in multiple times (naturally, with different names along the different paths as guaranteed by the action part of the rule, which concatenates the appropriate association names).

The rule is applied to each path once, and assumes that no circular associations exist between non-persistent classes. In this case, the termination of the

rule is guaranteed by the negative application pattern attrExists, which prohibits the application of the rule twice to the same Assoc - Association pair defined by the from and via edges in the helper structure of Assoc.

Finally, note that the merging-in of attributes can be treated in a similar way, although some additional post-processing is required to set primary keys properly to fresh attributes.

The transformation steps up to know generated helper structures (like Assoc, Attr or anc) in the source UML language only (layers 0–2 in the attributed graph transformation case). Now we discuss a major step taken from the main model transformation (layers 3–4), namely, how to generate columns and foreign keys from the helpers Assoc and Attr.

Generation of columns and primary keys. We assume that the model transformation has completed steps up to layer 3, that is, all persistent classes are transformed into corresponding database tables.

The requirements state that associations and attributes leading to a persistent class should be handled similarly. For that purpose, we extend the source meta-model with the notion of a Property, which is a common superclass of Attr and Assoc. Furthermore, we introduce a predefined graph pattern attrOrAssoc (depicted in Fig. 26), which can be interpreted as a hyperedge leading between the source and target class of an Assoc or Attr object.

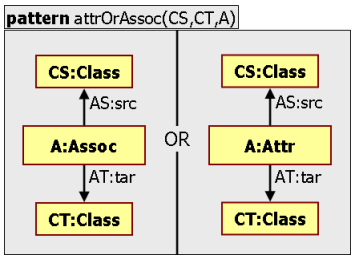

Fig. 26. Predefined helper graph patterns

Afterwards, we can express in a single GT rule (see rule assocOrAttrPersClassR in Fig. 27) that a foreign key should be generated for each attribute or association leading to a persistent class (which is already transformed into a table) by using this pattern. During this rule application, VIATRA calls the predefined pattern attrOrAssoc using the find construct.

Note that the action part of this rule contains a call to the GT rule addFKeyColsR (see Fig. 28) to copy all primary key columns from the target table T2 to the source table T1 for each matching of rule assocOrAttrPersClassR.

Furthermore, Property P and Column Cl2 are only required in the postcondition so that the action part could access their attributes when calculating the attributes of Column Cl1 from the attribute values of this nodes.

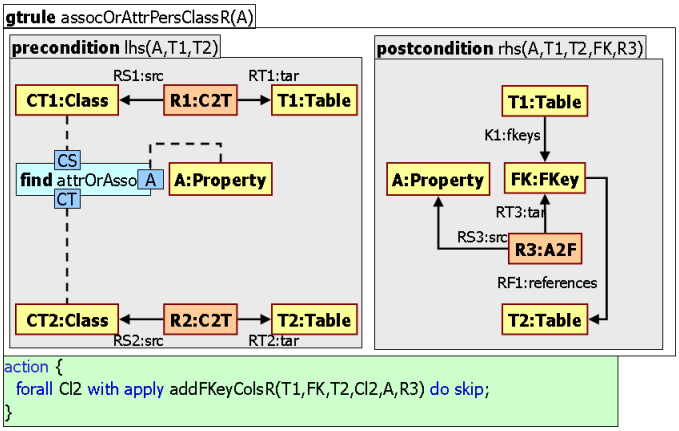

Fig. 27. Generating foreign keys for merged-in attributes and associations

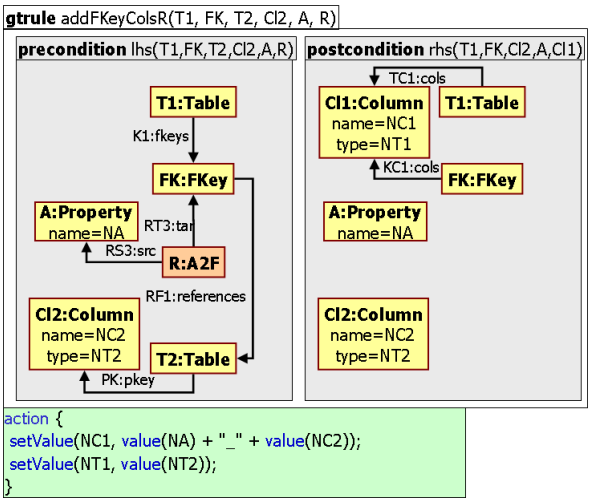

Fig. 28. Generating columns for foreign keys

For this purpose, we can pass the matched values of elements in the precondition (LHS) pattern (such as table T2) or the generated elements for elements in the postcondition (RHS) pattern (such as foreign key FK). Furthermore, we prescribe that rule addFKeyColsR should be applied to all matching (primary key) columns of table T2 by using the forall construct iterating over the values that can be taken by a fresh variable Cl2. Without using this forall, rule addFKeyColsR would be applied only on a single matching of Cl2.

Formally, the possible values of this variable are passed as parameters to the GT rule addFKeyColsR, but for efficiency reasons, VIATRA applies lazy evaluation, and thus enumerates the possible values of Cl2 only during the pattern

matching phase of this second rule (and *not* when calling this rule in the action part of GT rule assocOrAttrPersClassR).

Comparison with the algebraic approach. Compared to the basic algebraic solution presented in Sec. 3, the VIATRA approach provides (i) a more expressive pattern language, (ii) complex control structures to improve the efficiency of transformations (e.g. parallel rule application), (iii) generic transformations to provide algorithmic-level reuse of solutions for common transformation problems, (iv) structuring mechanisms for modularity achieved by pattern and rule parameters.

5.3 Tool Support: VIATRA2

The VIATRA approach is currently supported by the VIATRA2 model transformation framework, which was used for the implementation of the rules above. VIATRA2 is written in Java and it is fully integrated into the Eclipse framework (in contrast to the previous Prolog-based VIATRA toolkit).

Execution of transformations. Transformations are primarily executed within the framework by using the VIATRA interpreter, which uses constraint satisfaction techniques for matching graph patterns (which is typically the most expensive step of a model interpreter).

Furthermore, transformations can be externalized by compiling transformations into platform-specific, stand-alone transformation plug-ins (transformers). These automatically generated transformers are traditional pieces of software which can be integrated into existing tools without integrating the entire VIATRA framework.

Extension mechanisms. VIATRA provides an extensible framework for handling a large variety of MDA models and transformations. Arbitrary model importers can be easily implemented using the simple meta-modelling API of VPM metamodelling core, and a large set of importer plug-ins is already available. Furthermore, VIATRA transformations may call external Java methods if necessary to integrate external tools into a single tool chain.

User interface. Currently, VIATRA provides a textual language for describing models, meta-models and transformations (called VTML and VTCL, respectively), and a graphical user interface for viewing and editing the VPM model space including the creation and deletion of various model elements (and traditional undo and copy operations). This GUI provides both an Eclipse GEF-based fully graphical view and a tree view of the model space. Support for a graphical transformation editor of GT and ASM descriptions (similarly to the figures in this section of the paper) is an ongoing activity.

6 A Model Transformation Approach Based on VMTS

6.1 The Approach

The Visual Modeling and Transformation System (VMTS) is an n-layer meta-modelling environment designed together with model transformation functionalities. VMTS offers the most widely accepted constructs to define visual languages: built-in relationships are available for containment hierarchy, attributes and inheritance. Inheritance support means the following: if one submits a query for the attributes and the edges of a node-based model element (e.g. class), then all the inherited attributes and edges are retrieved besides the ones defined for the given element.

VMTS is also a model transformation system, which uses graph transformation techniques as the underlying formal mechanism. The general overview of the transformation process is illustrated in Figure 29. Since the transformation system is integrated in the meta-modelling environment, the meta-models of the input and output models are assumed to be available besides the obvious input model. The meta-models can be considered to be analogous to type graphs discussed in Section 3. Furthermore, the transformation engine takes the transformation description, which consists of a rule specification and a control flow part.

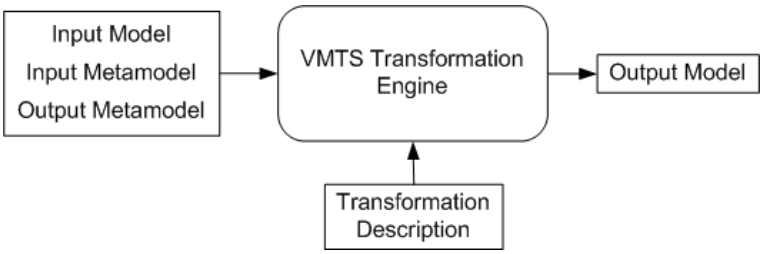

Fig. 29. Overview of the transformation process

The transformation rules used by VMTS are similar to the graph transformation rules: they contain a left hand side (LHS) and a right hand side (RHS). Also, a match of the LHS must be found in the model to which the rule is applied. As opposed to the graph transformation rules, not an isomorphic subgraph of the LHS must be found, but an instantiation of it, since it is defined in terms of the meta-model elements. This technique facilitates a natural representation for multiplicities, multi-objects and assignments of OCL constraints to the rules as well as a syntax close to the UML notation.

It is worth noting that VMTS facilitates a refined description of the transformation steps. When the transformation is performed, the changes are specified by the RHS and *internal causality* relationships defined between the LHS and the RHS elements of a transformation rule. Internal causalities can express the

modification or removal of an LHS element, and the creation of an RHS element. XSLT scripts can access the attributes of the objects matched to the LHS elements, and produce a set of attributes for the RHS element to which the causality points. Therefore, it is not true that if a rule element appearing in the LHS but not in RHS is going to be deleted.

On the rewriting level, VMTS follows the DPO rules: the elements to be deleted are removed first, then the elements to be added are glued to the model. Moreover, the DPO gluing conditions are enforced by the tool.

The VMTS Control Flow Language (VCFL) is a stereotyped UML activity diagram, where the activities are provided with the stereotype $<< TransformationRule >>$. The decisions contain OCL constraints to decide which branch must be passed the control. The OCL expressions may contain system variables, such as *SystemLastRuleSucceed*,which is true if the last rule has been executed successfully, false otherwise. VCFL allows specifying external causalities, which are a means of parameter passing between the rules. External causalities can be defined between the RHS of a rule $R1$ and the LHS of a rule $R2$, where there must exist a transition from $R1$ to $R2$ in the control flow. If the control is passed to this transition, the model objects matched to the source of the causality are automatically associated with the target of the external causality. The matching algorithm considers these LHS objects as already bound.

In the next section we show how these features can be applied to the common example already presented.

6.2 The Solution

In this section a solution supported by VMTS is given. We concentrate on the additional features of the basic solution elaborated in Section 3. The complete solution of the case study can be found in [26].

The VMTS uses a simplified class diagram to describe the meta-models. The subtle differences compared to Figure 4 are the following. (i) We specify two different meta-models, because we can use helper nodes, which can connect two nodes with any type. These helper nodes can also contain attributes. In accordance with the triple graph grammar method applied by ATOM^3, we create a $<< SystemAtom >>$ node with two edges pointing to the two connected objects. However, for the sake of simplicity, this construct is referred to as helper nodes. (ii) We specify multiplicities on the association ends, because it is required by our meta-modelling language.

In most cases we stick to the following principles. If the transformation is not supposed to modify the input model, it is not modified. In our example this is the case, because the task is to create an output model, namely, the database representation. By the end of the transformation, all helper nodes must be deleted and restored, except the ones used for tracing.

The VCFL representation of the transformation control flow is depicted in Figure 30.

The control flow can be divided into four parts according to the goal of the units. (i) The large loop on the top (*CreateTable*, *CreateParentClassHelper*,

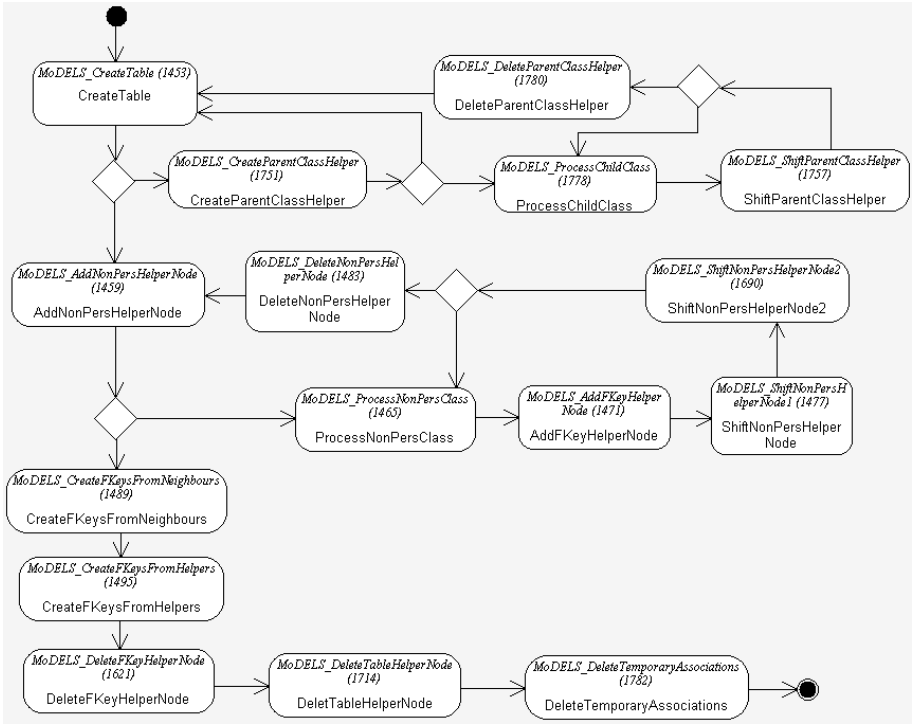

Fig. 30. Overview of the transformation process

ProcessChildClass, *ShiftParentClassHelper*, *DeleteParentClassHelper*) is responsible for treating the inheritance-related issues. (ii) The large loop in the middle (*AddNonPersHelperNode*, *ProcessNonPersClass*, *AddFkeyHelperNode*, *ShiftNonPersHelperNode*, *ShiftNonPersHelperNode2*, *DeleteNonPersHelperNode*) is devoted to process the chains of non-persistent classes. (iii) The last steps perform the computation of the foreign keys, and (iv) removing helper nodes.

A major challenge is to compute the "transitive closure" of the inheritance chain and the attribute change. This is a common problem of the inheritance hierarchy and the transformation of the non-persistent chains. In case of the inheritance support, we pull up all the attributes and associations from the hierarchy in the root base class if it is persistent. When it comes to non-persistent chains, the attributes and primary keys from the non-persistent classes must be incorporated into the persistent class to which the chain is attached. Because of the similarities, the attribute chains are described here in detail, the rules for the inheritance chain can be found in [26].

The first step (*CreateTable*, Figure 31) is equivalent with the "*class2table*" rule described earlier. This rule is applied exhaustively, thus, at the end each primary class has a table in the output model associated with it through a

helper node. In case of inheritance hierarchies only the topmost base class is regarded.

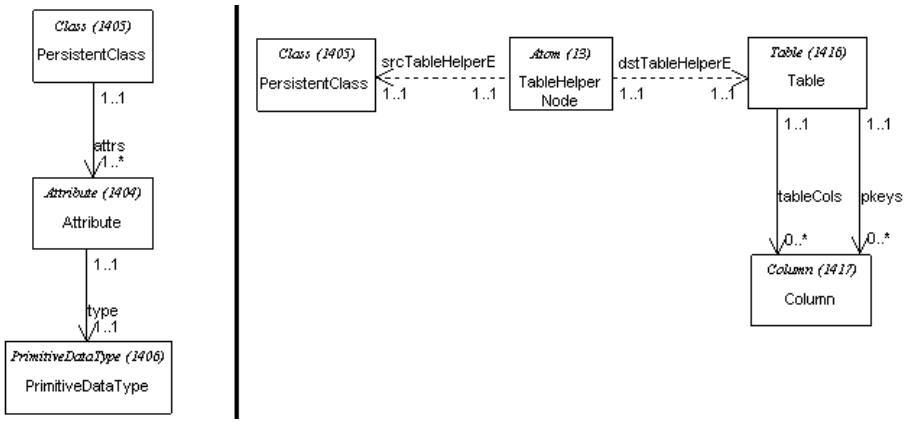

Fig. 31. The rule *CreateTable*

Our interpretation of the case study is the following. If an association from a persistent class PC points to a directed chain of non-persistent classes, then all the attributes of the non-persistent classes should be mapped into the table created from the persistent class as columns. Moreover, if an association points to another persistent class PCF from a class in the non-persistent chain, it should be referenced as a foreign key in the table created for the class PC. In our solution, a helper node is used for both, the transitive closure and the primary key management.

VMTS uses four kinds of helper nodes in the transformation. (i) *ParentClassHelperNode* is devoted to the creation of the transitive closure upwards in the inheritance hierarchy. (ii) The *ClassTableHelperNode* is used to connect persistent classes to their created tables. (iii) The *FkeyHelperNode* helper node denotes that a non-persistent class NP has a foreign key to a persistent class PCF. When the non-persistent class is mapped to a referencing persistent class PC, this foreign key must be included in the table of PC. In this case our solution creates a directed helper node from the table of PC to PCF. (iv)The *NonPersHelperNode* points to the actual non-persistent classes in the current chain to help in the computation of the transitive closure. Moreover, temporary associations are created in the base classes of the persistent hierarchies, which are denoted with an attribute. These nodes are deleted when they are not necessary anymore.

In accordance with these guidelines, the rule *AddNonPersHelperNode* selects a persistent class and assigns a helper node *NonPersHelperNode* to this class, pointing to all the adjacent non-persistent classes (Figure 32).

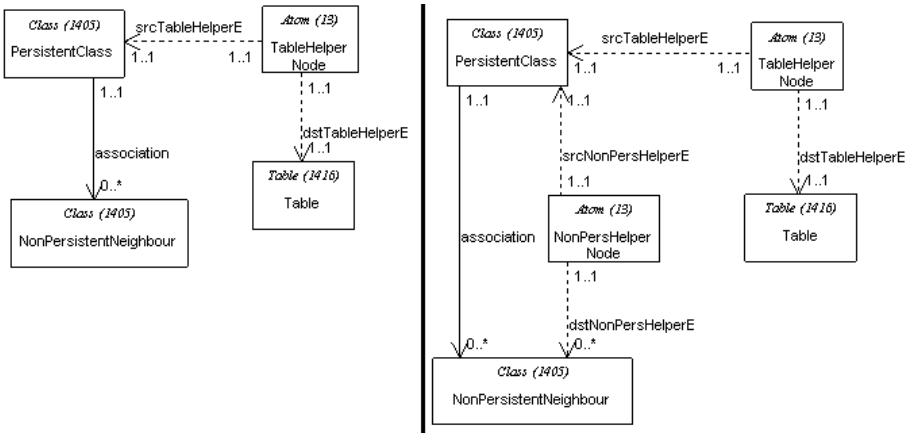

Fig. 32. The rule *AddNonPersHelperNode*

Both sides of the rule uses a formalism similar to that of the UML class diagram. The same instantiation rules apply to the LHS as to the UML class diagram, with two exceptions. (i) An association with the multiplicity $*$ matches all the edges of the appropriate type in a given position. (ii) A type can appear more than once in the rules. In our case both the *PersistentClass* and the *NonPersistentClass* are classes. These types are processed by the matching algorithm as they had been of different types. However, care is taken that the same model element cannot be matched to two different LHS objects. In our example this is impossible, since there are different constraints assigned to them: *PersistentClass* should be persistent and should not be processed yet. The *NonPersistentClass* should be non- persistent. These constraints are expressed in OCL, which is a natural construct in a UML class diagram environment. In our experience, the rule specification is closer to the intuition if it allows processing all neighbors of a model node. In VMTS, association multiplicities can be assigned to the rules, which are instantiated in the same way as in the UML class diagram. Thus, the rule can match all the non-persistent neighbours of a persistent class.

The interpretation of the undefined $(*)$ multiplicity is not evident on the right hand side. In VMTS, this value is determined by the result of the attribute transformation. The advantage of this approach is the flexibility that an attribute can influence the topology. The drawback is that the transformation cannot be executed without the attribute transformations: if there is at least one $(*)$ star multiplicity assigned to an association of RHS, the attribute transformation must be specified. Since VMTS is intended to be a model transformation system, the attribute transformation is usually available. Thus, this is not a serious limitation of the tool.

This rule also sets the *isProcessed* attribute of the actual persistent class to true. The attribute transformation is performed by XSL scripts. These scripts

are assigned to the internal causalities. In the rule *AddNonPersHelperNode*, there are three *Create* causalities in the rule (Table 6.2).

Causality Type	**Source**	**Target**
Create	association	dstNonPersHelperE
Create	association	NonPersHelperNode
Create	association	srcNonPersHelperE
Identity	PersistentClass	PersistentClass
Identity	Table	Table
Identity	NonPersistentNeighbor	NonPersistentNeighbor
Identity	TableHelperNode	TableHelperNode

Table 1. Causality table for the rule *AddNonPersHelperNode*

The causality, which starts from the association and points to the helper node *NonPersHelperNode*, takes the target role name of the *association* and copies it to the *namePrefix* attribute of the helper node. Since the attribute transformation computes a helper node attribute set for each matched *association*, the $*$ multiplicity of the newly created nodes are substituted by the actual number of *association* relationships in the matched model part.

The rule *ProcessNonPersClass* takes the persistent class as well as its attached table along with all the non-persistent classes adjacent via the helper nodes (Figure 33). The attributes of the matched non-persistent classes are transformed to the columns of the assigned table. The primary key nodes are also added.

The LHS matches all the non-persistent classes, to which a *NonPersHelperNode* points, along with their attributes and types. Then columns and primary key relationships are assigned to the table of the actual persistent class. Since the multiplicities in the RHS are not specified, the attribute transformation will determine the actual multiplicities. The number of the newly created columns is the same as the number of the attributes of the non-persistent classes. The primary key relationship is created for the primary key attributes only. The relative complex processing step of all the current non-persistent classes can be expressed by only one rule. This illustrates the advantages of the VMTS approach: although it takes more time to specify the rules, a relatively complex concern can be expressed by a rule in a straightforward and intuitive way. The causalities of the rule can be found in Table 6.2.

At the current stage of the transformation, there is only one persistent class, which is connected to a *NonPersHelperNode*. Thus, when the control is passed to the rule *ProcessNonPersClass* from the rule *AddNonPersHelperNode*, the *ProcessNonPersClass* rule must find this unique place in the model graph. This is a correct solution, but it can be optimized further in VMTS. The appropriate parameter passing constructs are called external causalities. The external causalities are attached to the transitions between the rules. In our example, the exter-

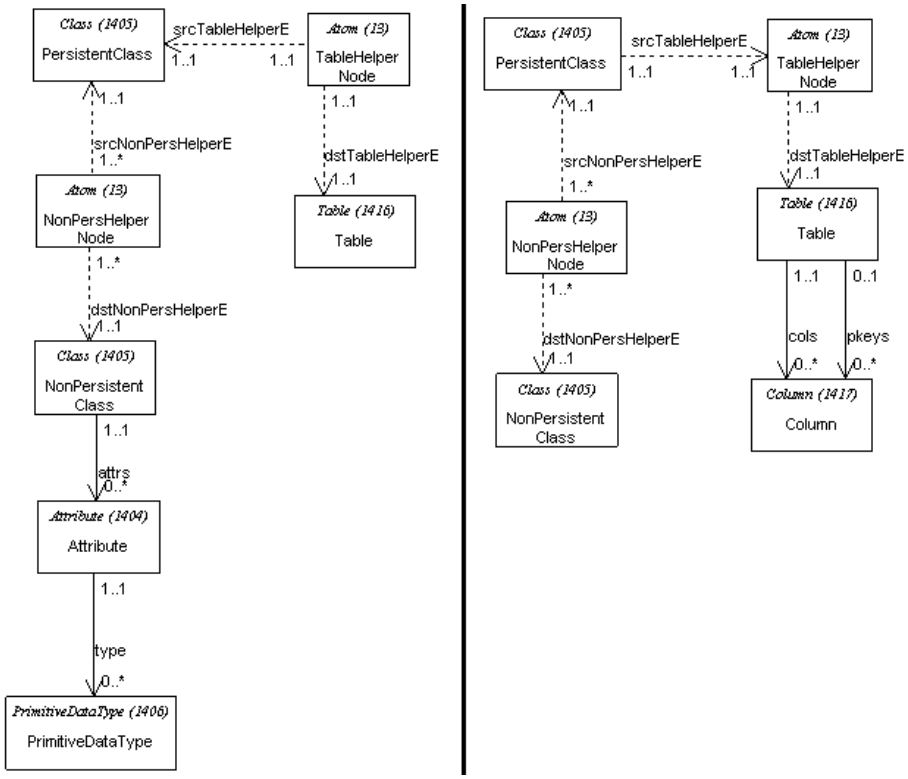

Fig. 33. The rule *ProcessNonPersClass*

Causality Type	Source	Target
Create	Attribute	Column
Create	attrs	cols
Create	attrs	pkeys
Identity	PersistentClass	PersistentClass
Identity	Table	Table
Identity	NonPersistentClass	NonPersistentClass
Identity	TableHelperNode	TableHelperNode
Identity	NonPersHelperNode	NonPersHelperNode

Table 2. Causality table for the rule *ProcessNonPersClass*

nal causalities are depicted in Figure 34. The persistent class, the related table, the table helper node, the non-persistent neighbor class and the non-persistent helper node are passed to the next rule. Thus, the *ProcessNonPersClass* transformation step must find the elements, which have been not concerned with the transformation so far, but for the other elements we reuse the elements already matched.

Name	Start Node	Destination Node
persClass	PersistentClass (1514)	PersistentClass (1522)
table	Table (1515)	Table (1523)
tableHelperNode	TableHelperNode (1728)	TableHelperNode (1732)
nonPersHelperNode	NonPersHelperNode (1730)	NonPersHelperNode (1733)
nonPersClass	NonPersistentNeighbour (1516)	NonPersistentClass (1524)

Fig. 34. The external causalities

The rule *AddFkeyHelperNode* selects all the non-persistent classes that are adjacent to the actual persistent class, having an edge pointing to another persistent class. The actual persistent class and the other persistent classes are connected with an edge *FkeyHelperNode*. We need this helper node, because it is possible that the other persistent classes are not yet processed, and can have additional primary keys originated from their non-persistent class chain. Hence, a foreign key cannot be created in this step, because the columns of the primary key that the foreign key references are not known yet.

The next two rules shift the helper nodes of type *NonPersHelperNode*. The attribute *namePrefix* of the helper node is concatenated with the target role name of the association along which the helper node is shifted. The concatenation is performed by the XSL script assigned to the internal causality. Recall, that the XSL script can access the attributes of the matched elements, and produce the attributes for the RHS element to which the causality point. Since, according to the DPO approach, the elements to be deleted are removed first, we cannot delete the previous helper node in this rule, because the value of the *namePrefix* attribute would be lost. Thus, this operation is performed by the rule *ShiftNonPersHelperNode2*. If the rules can successfully shift the helper nodes, the new non-persistent classes are processed. If the shifting rules cannot be applied, then the end of the non-persistent chain has been reached. Then the helper nodes are removed, and the rule *AddNonPersHelperNode* attempts to find a new actual persistent class. Since there is a constraint on the *isProcessed* attribute, the same persistent class cannot be processed twice. If this rule does not match, then all non-persistent chains are processed. Since we have all the columns and primary keys of the tables, the foreign keys can be constructed. It is done in two steps: (i) the adjacent persistent classes are considered and (ii) the helper nodes of type *FkeyHelperNode* are processed. The last three rules remove the remaining instances of the helper nodes.

6.3 Tool Support

The solution presented above has been specified and implemented within the Visual Modeling and Transformation System. The tool has successfully been applied in industrial applications such as that elaborated in [17].

The rule execution in VMTS is specified by VCFL, the pattern matching algorithm is MetaVF2 with heuristics [18]. The constraints given in OCL are

compiled into a .NET assembly, which is invoked by the transformation process. In order to obtain a more expressive constraint treatment, aspect-oriented techniques are used to eliminate the constraints that crosscut the transformation.

Import and export functions in VMTS are realized by Traversing Model Processors (TMP). VMTS offers a TMP interface, where the model elements appear as regular objects in a programming language, and traversing classes are also provided by the framework. The types of these objects are obtained from the meta-model.

As far as the user interface is concerned, VMTS incorporates VPF, which is a class library that supplies: (i) built-in base classes for the general presentation facilities of shapes (nodes) and lines (edges). (ii) Automatic event handling for the common functionalities such as resizing, moving and selecting model elements. (iii) Automatic serialization for the properties of the model elements. (iv) Sophisticated presentation of attributes, model structure, visualization information and editing features.

7 Comparison

Within this section, we compare the different graph transformation approaches to perform the sample model transformation. Each approach comes along with an accompanying tool. So we compare the corresponding tools along main model transformation facilities.

Since not only the approaches, but also the presented solutions differ, we also compare those. Be aware that we kept the original meta models for class models on one hand and for relational data base models on the other hand. But the additional helper structures, only transiently used for model transformations, can differ from each other.

Additionally, we compare the graph transformation approaches with the *Query/View/Transformation* language [22], since is supposed to become the OMG standard language for model transformation. A QVT solution for the basic version of the model transformation example can be found in the appendix.

A summary of the comparison is given in Fig. 35.

7.1 Comparison of Graph Transformation Approaches and QVT

Typing Information. In all approaches considered, the typing information is given by an attributed type graph or meta-model which contains the structural information (typing of entities and attributes), inheritance concepts and multiplicity constraints. In AToM^3, additional constraints can be expressed in Python, while AGG allows the formulation of graph constraints. In QVT, typing information is provided from the source and target meta-models for model transformation (see for example Fig. 1 and 2 in [4]) whereas in the graph transformation approach the model transformation type graph T consists of the source type graph T_S, the target type graph T_T and additional reference nodes and edges

Comparison of Model Transformation Approaches					
	AGG	AToM3	VIATRA2	VMTS	QVT (Core)
Type information	Type graphs with multiplicities	UML-like meta models	UML-like meta models	UML-like meta models	MOF metamodels
Instances	Typed attributed graphs	Models	Models	(instance) models	MOF models
Pre-conditions	LHS, application conditions (Java for attr.) and gluing condition	LHS, application conditions (Python for attr.), and gluing condition	LHS, recursive graph patterns, attribute conditions	OCL, LHS	OCL, patterns
Post-conditions	Post appl. conditions	Python code	graph patterns with attribute conditions	OCL	OCL, patterns
Actions	Deletion in L -K, creation in R-K, attribute computations in Java	Deletion in L – K, creation in R – K, attribute computation in Python	Deletion & creation based on R-L, attribute computations in ASM	Causalities based on LHS -> RHS	Patterns of enforced domains
Control	Non-deterministic rule and match selection, layers	Non-deterministic rule and match selection, parallel execution of rules, priorities	Abstract state machine (ASM) rules	Stereotyped activity diagrams	Context (when) and post-effect (where) clauses
Correctness	Critical pair analysis, termination, preservation of graph constraints	Correct typing of the target model	Correct typing of the target model	Preservation of OCL constraints	Patterns of checked domains
Additional features		Potentially incremental model transformation	Generic and meta transformations, Native (Java) transformations		Incremental model transformation
Comparison of Model Transformation Solutions					
Persistent classes: attributes of primitive data type	Columns of corr. tables	Columns of corr. tables	Columns of corr. tables	Columns of corr. tables	Columns of corr. tables
persistent classes: attributes of persistent classes	Columns of corr. tables – foreign keys	Columns of corr. tables – foreign keys	Columns of corr. tables – foreign keys	Columns of corr. tables -foreign keys	Columns of corr. tables – foreign keys
Inheritance: transitive closure	Additional ancestor helper edges between classes	Additional ancestor helper edges between classes and tables	Additional ancestor helper edges between classes and tables	Additional ancestor helper edges between classes	
Inheritance: attributes and associations of subclasses	Collection of additional attributes and associations in topmost classes	Incremental addition of attr. and assoc. to the table corresponding to the topmost class	Collection of additional attributes and associations in topmost classes		
Non-persistent classes: attributes and associations of non-persitent classes	Collection of additional attributes and associations in persistent class	1. treatment as persistent classes 2. copying to tables of persistent classes 3. deletion of tables of non-persistent classes	Collection of additional attributes and associations in persistent class	Coll. of additional attributes and assoc. in the corr. tables of the persistent classes	
Deletion of source model	Possible by deletion rules	Not considered	Not considered	Not considered	
Deletion of helper structure	Possible by deletion rules	Not considered	Not considered	By deletion rules	
Comparison of Model Transformation Tools					
Editors	Visual editors for graphs, type graphs, and rules	Visual editors for models, meta models, and rules	Visual editors for models, meta models, and textual language for rules	Adaptive modeler containing editors for meta models, models, and rules	
Simulation	Visual simulation of transformation seq., graph shown after each step	Visual simulation of transformation seq., graph shown after each step, timing of execution	Visualization of the result model	Visualization of the result model	
Compilation	None	None	Standalone Java code	None	
Debugging	Manual selection of rule and match possible	Manual selection of a match possible	Log window for model transformations	Log window for model transformations	
Validation	Critical pair analysis, termination, preservation of graph constraints	Correct typing of the target model	Correct typing + Preservation of graph patterns	Preservation of OCL constraints	

Fig. 35. Comparison of graph transformation approaches and QVT, solution and tools

needed by the model transformation rules (so-called helper structure). See Fig. 3.

Instances. Once a meta-model is defined, instances of this meta-model, i.e. models which conform to this meta-model can be created. This type-instance relation can be generalized as done in the meta-modelling approach where meta-models are instances of meta-meta-models, and so on.

Pre-conditions. The main precondition for rule application is the LHS of the rule. Furthermore, additional application conditions could be added. Rules also have additional attribute conditions which are expressed in Java, Python, or

OCL. Moreover in some approaches, the gluing condition is evaluated when the rule is going to be applied. In QVT pre-dependencies of a *relation* are defined in a pre-condition starting with the keyword *when*.

Post-conditions. Application conditions which are checked after a rule application are supported by each of the approaches in a similar form as pre-conditions. But the semantics can differ: While in AToM^3 the condition is checked only, the other approaches provide a rollback mechanism if the condition is not met. In QVT, post-dependencies of a *relation* are defined in the post-condition starting with the keyword *where*.

Actions. The actions of a rule can comprise deletion and creation of model entities as well as attribute modification. The latter is specified in Java, Python, ASM, and OCL, dependent on the approach. In addition, pre- and post-actions to be executed before and after the rule application, can be specified in AToM^3. In QVT, actions are formulated by enforcements. Attribute computations can be found in *where*-clauses.

Control. A completely uncontrolled rule application allows non-deterministic rule and match selection at any step of the transformation. To increase the usability and efficiency of graph transformation a variety of control concepts for rule and match selection have been considered. Some of them are used with the approaches presented: AGG uses rule layers, AToM^3 supports priorities for rules, VIATRA2 supports ASMs and VMTS offers stereotyped activity diagrams. In QVT *top relations* are applied directly whereas (more elementary) *relations* are called from *top relations*.

Correctness. Here mainly the preservation of constraints is supported. Moreover, AGG offers validation techniques to find conflicts between rules and to check termination criteria. Using triple graph grammars in AToM^3 supports checking the correct typing of the target model, as the auxiliary elements are kept in the correspondence graph.

Additional features. AToM^3 allows for some incremental transformations. Since the source elements and the correspondence graph in between the source and target are not deleted, there is the following option. If the user modifies the source model, the grammar can be executed again to update the target model. In this case, the transformation is incremental, but only if the user adds new elements in the class diagram. If the user deletes some class diagram elements, we would have to delete the associated relational model and start again the transformation. This could be solved by adding some rules dealing with the deletion of class diagram elements. In QVT, *relations* between domain models (instead of rules) are the basis for incremental model transformations.

VIATRA supports generic and meta transformations to provide reusable transformation libraries applicable to different meta-models. Furthermore, VIATRA provides language constructs to "officially" integrate native Java method calls in transformations, which is similar to the black-box (external) transformation concept of QVT.

7.2 Comparison of Model Transformation Solutions

Persistent classes: attributes of primitive data type. In all solutions a column in the corresponding table is added and in some solutions, correspondence relations are kept as helper structures.
Persistent classes: attributes of persistent classes. In all solutions foreign keys are created and in some solutions, correspondence relations are kept as helper structures.
Inheritance: transitive closure. In all solutions additional helper edges between classes are created to store the transitive closure of ancestors. In AToM^3, helper edges between the subclasses and the table associated with the top class are inserted.
Inheritance: attributes and associations of subclasses. In most solutions the attributes and associations of subclasses are collected in the topmost classes first and translated to columns afterwards, while in the AToM^3 solution, additional columns (of the topmost class' table) are added for each attribute of the subclasses incrementally.
Non-persistent classes: attributes and associations of non-persistent classes. In most of the solutions, attributes and associations of connected non-persistent classes are collected in referencing persistent classes first and then translated to corresponding columns. In the AToM^3 solution, tables are created for non-persistent classes (together with corresponding columns) first. Then, the columns are copied into the tables of persistent classes. Primary keys are updated accordingly. Finally, tables of non-persistent classes are deleted.
Deletion of source model. In AToM^3, VIATRA2, and VMTS deletion of the source model is not considered at all, while in AGG it can be done by a set of deletion rules, applied in a new rule layer.
Deletion of helper structure. In AToM^3 and VIATRA2 deletion of the helper structure is not considered at all, while in AGG it can be done and in VMTS it is always done, by a set of deletion rules, appropriately controlled.

7.3 Comparison of Accompanying Tools

Editors. All tools support the editing of meta-models, models and rules by visual editors. While AGG, VIATRA2 and VMTS concentrate on the abstract syntax of model transformations, AToM^3 is oriented to the definition of Domain Specific Languages, where the concrete syntax has also to be defined. Furthermore, it is possible to open several meta-models at the same time which is quite useful when implementing triple graph grammars.
Simulation. All tools can simulate transformations of concrete models. While VIATRA2 and VMTS concentrate on the visualization of the result model, AGG and AToM^3 also support the visualization of the model transformation sequences themselves, i.e. intermediate models. Moreover, executing the rules in AToM^3 allows for the visual animation of the rule applications, e.g. a timing information can be assigned to rules, so that every rule takes a certain amount of time to be executed.

Compilation. Most of the tools do not yet support the compilation of model transformations to some programming language. But VIATRA2 already supports a compilation to standalone Java code.

Debugging. All tools support some kind of logging for model transformations. These logs usually contain the rules applied, bindings of variables, etc. In addition, AGG supports the manual selection of rules and interactive setting of partial matches which might be completed automatically. In AToM^3, the user also can select manually the match where the rule has to be applied.

Validation. AGG, VIATRA2 and VMTS support some validation tools which check if constraints are preserved by models. In addition, type checking is supported. Moreover, AGG offers validation techniques to find conflicts between rules and to check termination criteria. AToM^3 offers a code generator for AGG to use AGG's validation tools within AToM^3.

8 Conclusions and Future Work

Summarizing the comparative study in this paper, we can notice that the graph transformation approaches and QVT share a number of commonalities. The typing concepts by type graphs or the almost equivalent concept of meta-models can be found in all approaches. Moreover, all transformation approaches considered are rule-based, even QVT with its concept of relations between domain models is very close. While the simple rule-based approach is unidirectional, triple graph grammars and QVT relations focus more on bidirectional or even multi-dimensional transformations. All approaches follow an idea of pre- and post-conditions expressed by some patterns, equipped with typical actions changing the models. Main differences can be found in the description of additional attributes using Java, Python, ASM or OCL as languages for attribute computations as well as conditions. Moreover, the control of rule applications ranges from pure rule-based approaches allowing a high degree of non-determinism, to rather controlled rule applications using mainly automata-based descriptions. In contrast, QVT allows the call of relations from others.

The solutions presented all have the same basic solution, but differ in the advanced solutions. They all use the same domain meta-models, but use different helper structures. These are mainly used for the advanced solution to collect transient information. While some approaches collect this data directly in the source models, others extend the relations between source and target models. These differences are not large and it is an obvious conjecture that the differences in the solution are approach-independent.

Each graph transformation approach comes along with a tool which show different specialities. While AGG focusses on validation issues, AToM^3 concentrates on incremental transformations taking also layouting of the target models into account. VIATRA2 and VMTS are both closer to the MOF approach and thus to QVT. They use graph transformation concepts to come up with MOF transformations. All tools come along with a number of exchange formats sup-

ported such as GXL, XMI, etc. In the case of AGG and VIATRA2, the internal transformation engine can be used by a Java API.

Open issues for all graph transformation approaches are elaborated concepts to compose transformations as well as a suitable exception handling. Moreover, further tools might come up with a compilation of model transformations to some standard programming language such as Java (as VIATRA2 already does).

This comparative study is a first effort to compare the graph transformation approaches presented. Another one, where AGG and VIATRA2 were also compared, has been given in [19]. This paper applies a general model transformation taxonomy to graph transformation approaches and tools. Considering both studies, future work is needed to intensify the comparison by investigating further tools and by adding benchmark tests like the one considered in this paper.

References

1. AGG Homepage. http://tfs.cs.tu-berlin.de/agg.
2. Reusable Idioms and Patterns in Graph Transformation Languages. Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G. 2004. *Proc. 2nd International Workshop on Graph Based Tools (GraBaTs 2004).* Satellite workshop of ICGT 2004, Rome, Italy, 2004.
3. R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria, editors, *Proceedings of FASE 2004*, pages 214–228, 2004.
4. Model Transformation in Practice Workshop Announcement. Bezivin, J., Rumpe, B., Schürr A., Tratt L. 2005. http://sosym.dcs.kcl.ac.uk/events/mtip/
5. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag (2003)
6. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Termination of High-Level Replacement Units with Application to Model Transformation. In M. Minas, editor, *Proc. Workshop on Visual Languages and Formal Methods*, 2004.
7. H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In *Proc. Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, pages 00–00. Springer Verlag, 2005.
8. Formal Integration of Inheritance with Typed Attributed Graph Transformation for Efficient VL Definition and Model Manipulation. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2005. *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05).*
9. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1. Foundations.* World Scientific.
10. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H. 2004. *Constraints and Application Conditions: From Graphs to High-Level Structures.* In Proc. ICGT'04 (Rome). LNCS 3256, pp.: 287-303. Springer.
11. H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proceedings of ICGT 2004*, volume 3256 of *LNCS*, pages 161–177. Springer, 2004.

12. Guerra, E., de Lara, J. 2004. *Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation*. In Proc. ICGT'04 (Rome). LNCS 3256, pp.: 54-69. Springer.
13. Guerra, E., Díaz, P., de Lara, J. 2005. *A Formal Approach to the Generation of Visual Language Environments Supporting Multiple Views.* Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE VL/HCC. Dallas.
14. H.-J. Kreowski and S. Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11:690–723, 1999.
15. de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* LNCS 2306, pp.: 174 - 188. Springer. See: `http://atom3.cs.mcgill.ca`
16. Levendovszky T., Lengyel L., Mezei G., Charaf H.: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, 2nd International Workshop on Graph Based Tools (GraBaTs); workshop at ICGT 2004, Rome, Italy, 2004.
17. Lengyel L, Levendovszky T, Mezei G, Forstner B, Charaf H: Metamodel-Based Model Transformation with Aspect-Oriented Constraints, Accepted to GraMoT'05 - International Workshop on Graph and Model Transformation,Tallinn, 2005
18. Levendovszky T, Lengyel L, Charaf H: A UML Class Diagram-Based Pattern Language for Model Transformation Systems, WSEAS Transactions on Computers, ISSN: 110-92750 Issue 2, Vol. 4, 2005 pp. 190-195
19. T. Mens, P. Van Gorp, D. Varro, and G. Karsai. Applying a Model TRansformation Taxonomy to Graph TRansformation Technology. In G. Karsai, and G. Taentzer, editors, *Proceedings of Graph and Model Transformation Workshop*, to appear in ENTCS, 2005.
20. Object Management Group, *OMG Meta Object Facility (MOF), Version 1.4* (12th June 2003), URL: `http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf` `access: 2004-06-28`
21. Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars.* In LNCS 903, pp.: 151-163. Springer.
22. Query/View/Transformation. QVT-Merge Group, version 2.0 (2005-03-02), 2005. http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf
23. Varró, D.: Automated Model Transformations for the Analysis of IT Systems. PhD thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems (2004)
24. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2** (2003) 187–210
25. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds.: Proc. UML 2004: 7th International Conference on the Unified Modeling Language. Volume 3273 of LNCS., Lisbon, Portugal, Springer (2004) 290–304
26. VMTS Web Site, `http://avalon.aut.bme.hu/~tihamer/research/vmts`

A A Basic Solution using QVT

In the appendix we present a basic solution for the model transformation example in [4] using the *Query/View/Transformation (QVT)* language derived from the UML to RDBMS example in the QVT specification [22].

The basic solution shows only some aspects of the QVT language. Please note, that in general model transformation in QVT is also possible with different constructions shown in the QVT specification [22].

```
transformation classToRdbms(classmodel:ClassModel, rdbms:RDBMS)
{
        key Table (name); // Table has unique name
        key Column (name); // Column has unique name
        key FKey (references); // FKey has unique reference to Table

        top relation ClassToTable // map each persistent class to a table
        {
                cn, prefix: String;
                checkonly domain classmodel c:Class {is_persistent=TRUE, name=cn};
                enforce domain rdbms t:Table {name=cn};
                where {
                        prefix = '';
                        AttributeToColumn(c, t, prefix);
                }
        }

        relation AttributeToColumn
        {
                checkonly domain classmodel c:Class {};
                enforce domain rdbms t:Table {};
                primitive domain prefix:String;
                where {
                        PrimitiveAttributeToColumn(c, t, prefix);
                        ComplexAttributeToColumn(c, t, prefix);
                        SuperAttributeToColumn(c, t, prefix);
                }
        }

        relation PrimitiveAttributeToColumn
        {
                an, pn, cn, sqltype: String;
                checkonly domain classmodel c:Class {attrs=a:Attribute {name=an,
                        type=p:PrimitiveDataType {name=pn}}};
                enforce domain rdbms t:Table {cols=cl:Column {name=cn,
                        type=pn}};
                primitive domain prefix:String;
                where {
                        cn = if (prefix = '') then an else prefix+'_'+an endif;
                }
        }

        relation ComplexAttributeToColumn
        {
                an, newPrefix: String;
                checkonly domain classmodel c:Class {attrs=a:Attribute {name=an,
                        type=tc:Class {}}};
                enforce domain rdbms t:Table {};
                primitive domain prefix:String;
                where {
                        newPrefix = prefix+'_'+an;
                        AttributeToColumn(tc, t, newPrefix);
                }
        }

        relation SuperAttributeToColumn
        {
                checkonly domain classmodel c:Class {general=sc:Class {}};
                enforce domain rdbms t:Table {};
                primitive domain prefix:String;
                where {
                        AttributeToColumn(sc, t, prefix);
                }
```

```
	}

	// map each association between persistent classes to a foreign key
	top relation AssocToFKey
	{
		srcTbl, destTbl: Table;
		pColumn: Column;
		an, scn, dcn, pcn, fcn, fct: String;
		checkonly domain classmodel a:Association {
			name=an,
			src=sc:Class {is_persistent=TRUE,name=scn},
			dest=dc:Class {is_persistent=TRUE,name=dcn}
		};
		enforce domain rdbms fk:FKey {
			cols=fc:Column {name=fcn,type=fct},
			references=destTbl
		};
		when { /* when refers to pre-condition */
			ClassToTable(sc, srcTbl);
			ClassToTable(dc, destTbl);
			pColumn = destTbl.pkey;
		}
		where {
			fcn = an+'_'+pcn;
			pcn = pColumn.name;
			fct = pColumn.type;
			srcTbl.fkeys = fk;
			srcTbl.cols = fc;
		}
	}

}
```