ELECTRONIC COMMUNICATIONS OF THE EASST

Volume 4

Proc. of 2nd International Workshop on Graph and Model Transformation 2006

ISSN 1863-2122

Realizing QVT with Graph Rewriting-Based Model Transformation

László Lengyel , Tihamér Levendovszky and Hassan Charaf

12 pages, 2006

Realizing QVT with Graph Rewriting-Based Model Transformation

László Lengyel *, Tihamér Levendovszky * and Hassan Charaf *
*Budapest University of Technology and Economics

Abstract. *Model-based development is an increasingly applied method in producing software artifacts that is driven by model transformation. For instance, OMG's Model-Driven Architecture as a model-based approach to software development facilitates the synthesis of application programs from models created using customized, domain-specific model processors. Meta Object Facility 2.0 Query/Views/Transformation (QVT) is the OMG's standard for specifying model queries, views, and transformations. Extensive research of graph transformation provides a strong formal background for model transformation. The main contribution of this paper is to show how high-level constraint constructs facilitate to realize transformations specified in QVT with metamodel-based model transformation. As a result we can reuse the graph transformation constructs, and its formal background, which facilitates to make QVT transformations validated.*

Keywords: Graph Rewriting, Model Transformation, QVT Realization, High-Level Constraints.

1 Introduction

OMG's Model-Driven Architecture (MDA) [OMDA] emphasizes the use of models at all stages of system development. It has placed model-based approaches to software development into focus. MDA offers a standardized framework to separate the essential, platform-independent information from the platform-dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM), one or more platform-specific models (PSM) including complete implementations, one on each platform that the application developer decides to support. The platform-independent artifacts are mainly written in UML and other software models containing enough specification to generate the platform-dependent artifacts automatically by model compilers.

Transformations appear in many different situations in a model-based development process. A few representative examples are as follows. (i) Refining the design to implementation; this is a basic case of PIM/PSM mapping. (ii) Aspect weaving; the integration of aspect models/code into functional artifacts is a transformation on the design. (iii) Analysis and verification; analysis algorithms can be expressed as transformations on the design.

One may conclude that transformations in general play an essential role in model-based development, thus, there is a need for highly configurable model transformation tools. These tools must make the model transformation flexible and expressive. Furthermore, they should support control flow, constraints, parameter passing between sequential rules, and conditional branching.

The Model-Driven Architecture offers a standard interface to implement model transformation tools. The transformation related part of MDA is the Query/View/Transformation (QVT) for MOF 2.0 [Que]. Three types of operations are provided: queries on models, views on metamodels and transformations on models. In model transformation area QVT is one possible solution for defining transformations. But there are several different approaches and languages that offer similar constructs with minor/major differences in features.

With the hope that QVT become a wide-spread standard for model transformation, the goal of the current work is to introduce the realization of QVT Relations with graph rewriting-based model transformation, namely, with the Visual Modeling and Transformation System (VMTS) approach [VMT]. VMTS supports validated online transformation, therefore, using the results presented in the current paper we can transform model transformations defined in QVT to VMTS constructs, which means that they can also be validated. Furthermore, the results can be applied to transformations that are specified using a language whose artifacts can be transformed to QVT language. This means that there exists an implemented transformation, which maps the transformation specifications to QVT transformations.

There are several graph rewriting-based model transformation tool (e.g. GReAT [KASS03], PROGRES [RS97], FUJABA [KNNZ00], VIATRA [VP03], AGG [Tae03], and AToM^3 [dLVA04]), therefore, the mapping presented in this paper facilitates the reuse of the transformations that supports QVT as a common platform.

If QVT can be realized by graph rewriting-based model transformation, there is a hope that the results originating from the strong background of graph transformation can be reused on the QVT level. Furthermore, high-level constraint constructs provided by VMTS [LLC05] facilitates the online validated model transformation that also can be applied in QVT domain.

The rest of this paper is organized as follows. Section 2 provides the background information including a graph rewriting-based model transformation system (Visual Modeling and Transformation System, VMTS). Using the features of this system, the principles of metamodel-based model transformation are presented. Section 3 gives an overview on OMG's QVT language. Section 4 discusses the relation between QVT and VMTS constructs. Section 5 presents the realization of the QVT constructs with VMTS approach, thus, with a graph rewriting-based model transformation framework. Finally, conclusions are provided.

2 Backgrounds

Graph rewriting [Roz97] [EEKR99] is a powerful technique for graph transformation with a strong mathematical background. The atoms of graph transformations are rewriting rules, each rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of LHS in the graph to witch the rule being applied, and replacing this subgraph with RHS.

VMTS supports editing models according to their metamodels, and allows specifying Object Constraint Language (OCL) constraints. Models are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel (”visual vocabulary”).

Also, VMTS is a model transformation system, which transforms models using graph rewriting techniques. Moreover, the tool facilitates the verification of the constraints specified in the transformation rule during the model transformation process.

In VMTS, LHS and RHS of the transformation rules are built from metamodel elements. This means that an instantiation of LHS must be found in the input graph instead of the isomorphic subgraph of LHS.

Rewriting rules can be made more relevant to software engineering models if the metamodel-based specification of the transformations allows assigning OCL constraints to the individual transformation rules. This technique facilitates a natural representation for multiplicities, multi-objects and assignments of OCL constraints to the rules with a syntax close to the UML notation.

VMTS facilitates a refined description of the transformation rules. 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 or Imperative OCL [Que] scripts can access to 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.

Classical graph grammars apply any production that is feasible. This technique is appropriate for generating and matching languages but model-to-model transformations often need to follow an algorithm that requires a stricter control over the execution sequence of the rules, with the additional benefit of making the implementation more efficient.

The VMTS approach is a visual approach, thus, it also uses graphical notation for control flow: stereotyped UML activity diagrams. VMTS Visual Control Flow Language (VCFL) is a visual language for controlled graph rewriting and transformation, which supports the following constructs: sequencing transformation rules, branching with OCL constraints, hierarchical rules, parallel execution of the rules, and iteration.

The VMTS transformation rules have two specific properties: *Exhaustive* and *MultipleMatch*. Applying a model transformation rule means finding a match of LHS in the input model and replacing this subgraph with RHS. An *exhaustive* transformation rule is executed repeatedly, as long as LHS of the rule can be matched to the input model. The *MultipleMatch* property of a rule allows that the matching process finds not only one but all occurrence of LHS in the input model, and the replacement is executed on all the found places.

The interface of the transformation rules allows the output of one rule to be the input of another rule (parameter passing), in a dataflow-like manner. In VCFL, this construction is referred to as *external causality*. An external causality creates a linkage between a node contained by RHS of the rule i and a node contained by LHS of the rule $i+1$. Since rule i provides partial match to rule $i+1$ this feature accelerates the matching and reduces the complexity.

VMTS has state-of-the-art mechanisms for validated model transformation, constraint management and control flow definition. The environment has several standalone algorithms and other solutions that make them efficient. Moreover, VMTS has a unique, aspect-oriented technique-based constraint management [VMT]. The constraint-driven branching mechanism of the VMTS is unique in the sense that the decision is made not only based on the actual state of the input model but using system variables (*SystemLastRuleSucceed*) as well. If a transformation rule fails, and the next element in the control flow is a decision object, then it could provide the next branch based on the constraints. This VMTS construct accelerates and makes the transformation more efficient, and the control flow model simpler, because, for example, there is no need to define test rules as in GReAT or PROGRES.

3 MOF 2.0 Query/Views/Transformation Overview

The QVT specification has a both declarative and imperative nature, with the declarative part split into a two-level architecture that forms the framework for the execution semantics of the imperative part [Que].

The layers of the declarative part are the following: (i) The user-friendly *Relations* metamodel and language which supports complex object pattern matching and object template creation. (ii) A *Core* metamodel and language is defined using minimal extensions to EMOF [MOF] and OCL [OOCL].

The Relations language supports complex object pattern matching, and implicitly creates trace classes and their instances to record what occurred during the execution of the transformation. Relations can assert that other relations also hold between particular model elements matched by their patterns.

The Core language supports pattern matching over a flat set of variables by checking conditions over those variables against a set of models. It treats all of the model elements of source, target and trace models symmetrically (Figure 1a). It is equally powerful to the Relations language, and, because of its relative simplicity, its semantics can be defined more simply, although transformation descriptions provided using the Core are therefore more verbose. In Relations, transformation classes (or trace classes) are not explicitly specified and used. Instead, a relation directly specifies the relationship that should hold between source and target domains. In Core, transformation classes and patterns over them are an essential part of the mapping specifications. A relation is an assertion of a relationship that exists between the source and target model elements, and a transformation class essentially serves to capture such assertions structurally.

The semantics of the Core language and the Relations language allow the following execution scenarios: (i) check-only transformations to verify that models are related in a specified way, (ii) single direction transformations, (iii) bi-directional transformations, (iv) the ability to establish relationships between pre-existing models, (v) incremental updates when a related model is changed after an initial execution, and finally, (vi) the ability to create or delete objects and values, while also being able to specify which objects and values must not be modified.

4 Relation between QVT and VMTS Constructs

This section summarizes the relation between the QVT and metamodel-based model transformation. Table 1 compares the model transformation related basic constructs of the QVT and the VMTS .

Figure 1b introduces the principle of the metamodel-based model transformation rules. The constructs that form a metamodel-based LHS specification are inheritance and multiplicity support (Figure 1b). In-

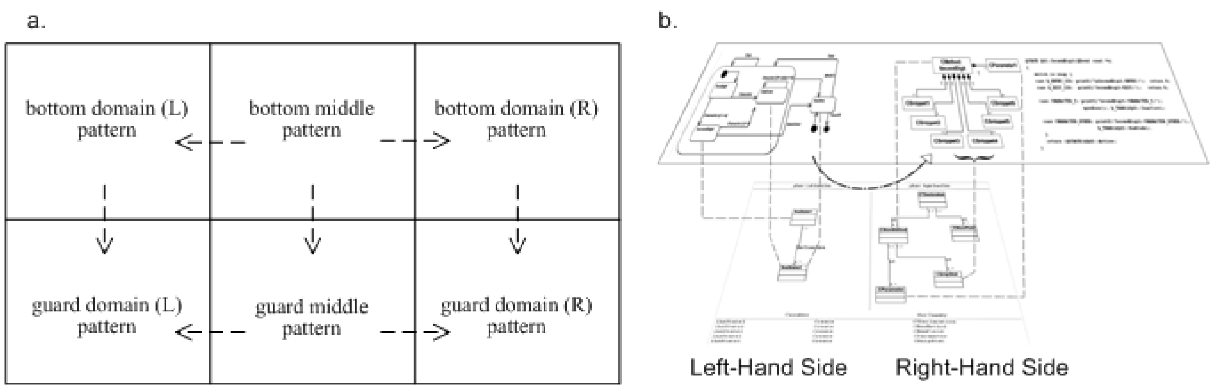


Figure 1: (a) QVT Core domain and pattern dependencies, (b) VMTS metamodel-based transformation rule with input and output models

	QVT Construct	VMTS Construct
Type information	MOF metamodels	metamodels - instantiation relation is based on the relation between MOF layers 0 and 1
Instances	MOF models	(instance) models
Preconditions	OCL, patterns	OCL, LHS
Postconditions	OCL, patterns	OCL, RHS
Actions	Patterns of enforced domains	Internal causalities based on LHS $\rightarrow$ RHS
Control	Context (when) and post-effect (where) clauses	Stereotyped activity diagrams (VCFL)
Correctness	Patterns of checked domains	Preservation of OCL constraints

Table 1: Comparison of QVT and VMTS constructs

heritance support is analogous to the natural type compatibility of object-oriented languages: the derived class can always be passed where the ancestor class is expected. This means that a class element in LHS always matches its descendant types in the input model. That facilitates generalization in the rules as well as abstract types. Multiplicity support is accomplished by allowing multiplicity values on the association ends. In VMTS, the match found for LHS is maximal in a sense that the actual matched multiplicity is the greatest possible value from the specified multiplicity interval, which does not contradict any other part of the match.

The execution of metamodel-based model transformation in VMTS is depicted in Figure 2. The figure describes that the transformation is specified by the VCFL control flow model that defines the exact execution order of the transformation rules. The input model is described by the input metamodel, and the output model by the output metamodel. Both input and output metamodels have an effect on the transformation.

LHS and RHS can use different metamodels. Transformation rules contain OCL constraints. The transformation uses matches found by the matching process and the compiled binary generated by the

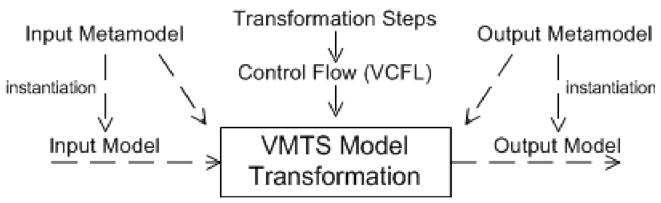


Figure 2: Principles of VMTS metamodel-based validated model transformation

OCL compiler to check the constraints on the matched parts of the input model. The transformation produces the transformation result if and only if a match satisfies the preconditions. Moreover, the rule is successful if and only if the transformation result satisfies the postconditions.

Transformations, Model Types and Mappings. Figure 1 presents the correspondence between QVT Core language transformation and VMTS metamodel-based transformation rules. To emphasize the similarity, we have swapped the Guard and Bottom rows in Figure 1a. In the Core language, a transformation is specified as a set of mappings that declare constraints that must hold between the model elements belonging to a set of candidate models and the trace model. The candidate models are named, and the types of elements that they can contain are restricted by a model type. Figure 1a depicts the structure of a QVT mapping with two domains, where each rectangle of a mapping represents a pattern. The columns are called areas. Each area consists of two patterns, the guard pattern and the bottom pattern. A mapping consists of one area for the trace (the middle area) and one area (a domain) for each model type. The domain areas consist of patterns that match the candidate models, the middle area consists of patterns that match the trace model.

A VMTS transformation is a VCFL model that is built from single metamodel-based model transformation rules. Transformation rules match the input models based on the defined structure, metatypes, and other constraints. The constraints are expressed in OCL. The transformation creates the output based on RHS and internal causalities.

Patterns and Binding. A QVT pattern is specified as a set of variables, predicates and assignments. Patterns can be matched and enforced. Matching a pattern can result in value bindings of the variables, and enforcing a pattern can result in model changes causing new value bindings for the variables during matching. In a mapping, the bottom patterns depend on guard patterns (in the same column) and middle patterns depend on domain patterns (in the same row).

In VMTS, the matched pattern corresponds to the match found for LHS of a rule, and the enforced pattern is the resulted model that is an instance of RHS. VMTS provides a unique constraint management mechanism that facilitates to reuse transformation rules with different constraint sets [VMT].

Guards and Checking. Guards of a mapping narrow the selection of model elements to be considered for the mapping. Matching the bottom patterns takes place in the context of a valid combination of valid bindings of all the guard patterns. In such a combination, for each dependency between two guard patterns, there must be exactly one dependency between two valid bindings of those two guard patterns.

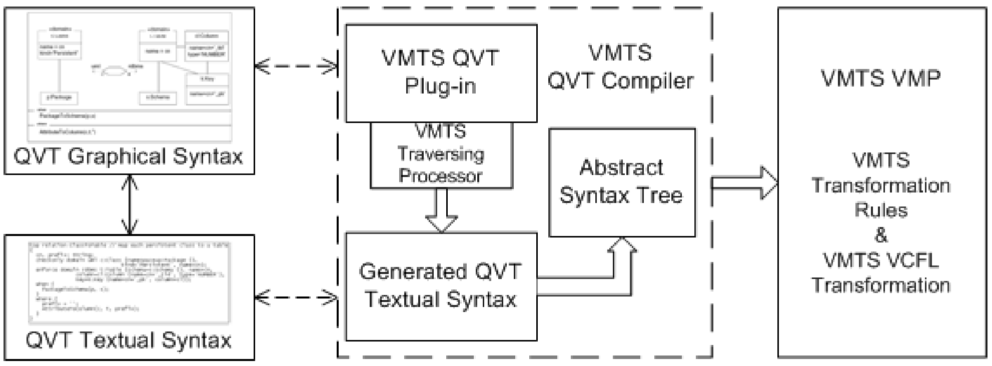

QVT Graphical Syntax
QVT Textual Syntax
VMTS QVT Plug-in
VMTS QVT Compiler
VMTS Traversing Processor
Abstract Syntax Tree
Generated QVT Textual Syntax
VMTS VMP
VMTS Transformation Rules & VMTS VCFL Transformation

Figure 3: Overview of the QVT realization with VMTS

Mappings can be checked, either as a part of a transformation execution in checking mode, or in the first rule of a transformation execution in enforcement mode. A transformation execution in checking mode will produce an error for each violation of a mapping constraint. A transformation execution in enforcement mode will enforce a repair for each violation of a mapping constraint.

In VMTS, guards are the metamodel-based model transformation rules. Checking can be expressed with transformation rules that contain only LHS. LHS expresses the required structure with metatypes and the necessary conditions defined by the propagated OCL constraints. If there is no proper match in the input model, then the rule fails.

Enforcement and Trace. At execution time, one model type of the transformation can be chosen as the enforcement direction. The target model and trace model may be changed to fulfil the constraints of the mappings. The models are only changed when the constraints of a mapping are not fulfilled. The changes will lead either to the creation of new valid bindings or the removal of existing valid bindings of the target bottom patterns and the trace bottom patterns to enforce the constraints of a mapping.

VMTS stores model elements (nodes and edges) in database tables. Each node is contained in the table NODE and each edge by the table EDGE. Each model related operation uses these tables to select, create or update elements. The VMTS database, among others, contains two tables (TRACE_NODE and TRACE_EDGE) in order to support tracing and to store the trace information separately from model elements. This means that trace information is created and used only by transformations and not by the presentation framework. Therefore, at modeling time trace information is not visible and does not confuse the source and target models.

VMTS trace objects are created based on the internal causalities of the transformation rules. Using the trace objects with the help of constraint management, VMTS supports model evolution. Once a relationship has been established between models by executing a transformation and creating trace objects, changes to a source model may be propagated to a target model by re-executing the transformation in the context of the trace, causing only the relevant target model elements to be changed, without modifying the rest of the model. Constraints and trace objects are used to ensure that the transformation rules generate the output only from those parts of the input model that have no generated output model or

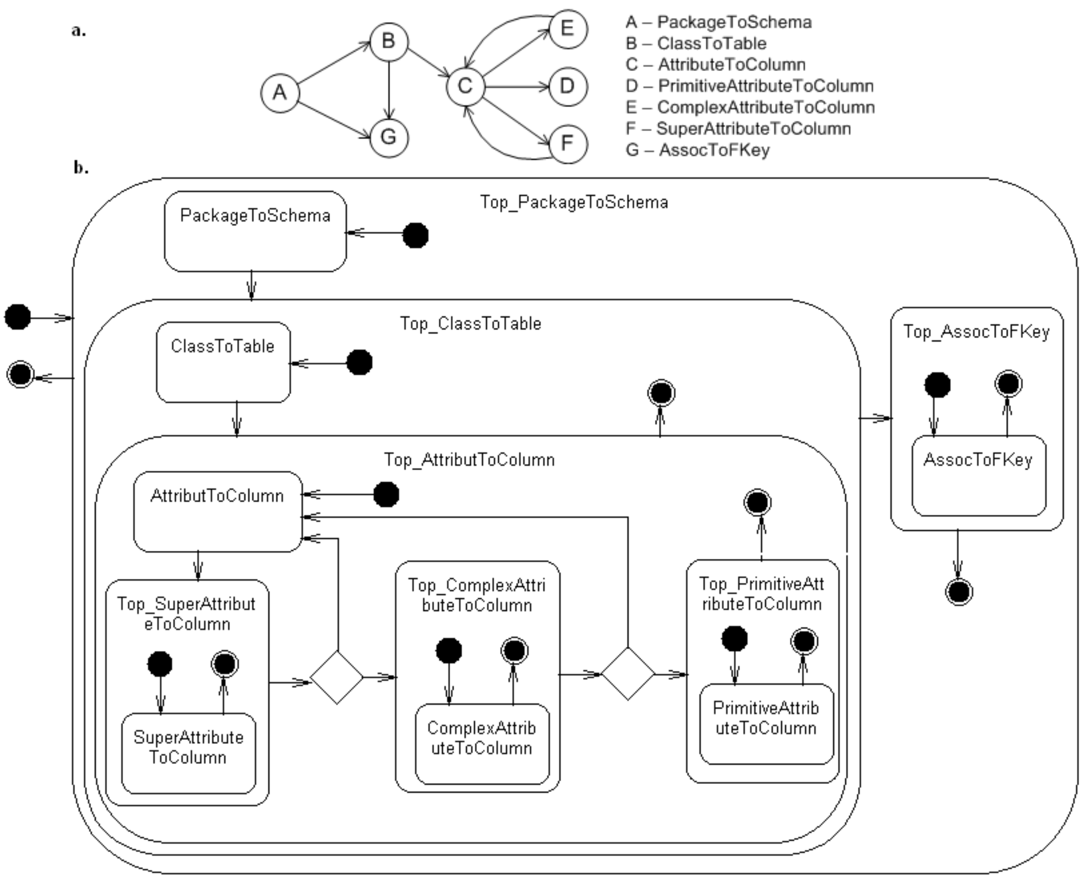
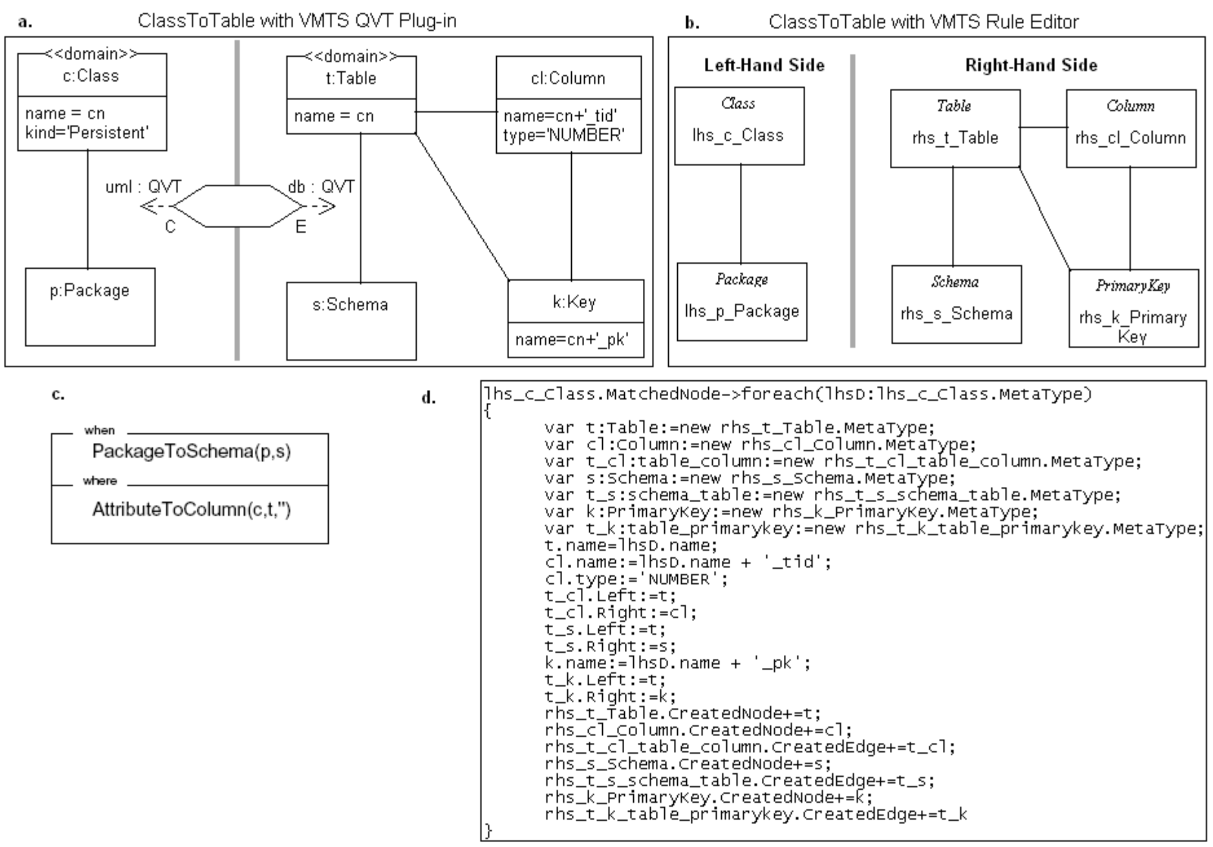


Figure 4: (a) Control Flow (CF) graph of the transformation *ClassToRDBMS* created by transformation *QVTConstructToVMTSConstruct*, (b) Generated VCFL model of the VMTS version of the transformation *ClassToRDBMS*

that have been modified since the last execution of the transformation. In the input model, each element has a *Modified* property that indicates whether it has been modified. It is updated and checked by the transformation process, and it is also corrected during any model modification. The trace information facilitates to decide if an input model element has generated target model elements or not.

It can be stated that design and dependency information must be preserved for software evolution in order to be automated. The MDA-based approaches define design information in high-level instances, while dependency information is provided in transformations. Therefore, the software evolution support can be solved with metamodel-based model transformations and refined with adequate constraint management.

Bi-directional transformations. The QVT operational mapping and black-box approaches restrict the scenarios by allowing specification of transformations in a single direction only. Bi-directional transfor-



```
lhs_c_Class.MatchedNode->foreach(lhsD:lhs_c_Class.MetaType)
{
        var t:Table:=new rhs_t_Table.MetaType;
        var cl:Column:=new rhs_cl_Column.MetaType;
        var t_cl:table_column:=new rhs_t_cl_table_column.MetaType;
        var s:Schema:=new rhs_s_Schema.MetaType;
        var t_s:schema_table:=new rhs_t_s_schema_table.MetaType;
        var k:PrimaryKey:=new rhs_k_PrimaryKey.MetaType;
        var t_k:table_primarykey:=new rhs_t_k_table_primarykey.MetaType;
        t.name=lhsD.name;
        cl.name:=lhsD.name + '_tid';
        cl.type:='NUMBER';
        t_cl.Left:=t;
        t_cl.Right:=cl;
        t_s.Left:=t;
        t_s.Right:=s;
        k.name:=lhsD.name + '_pk';
        t_k.Left:=t;
        t_k.Right:=k;
        rhs_t_Table.CreatedNode+=t;
        rhs_cl_Column.CreatedNode+=cl;
        rhs_t_cl_table_column.CreatedEdge+=t_cl;
        rhs_s_Schema.CreatedNode+=s;
        rhs_t_s_schema_table.CreatedEdge+=t_s;
        rhs_k_PrimaryKey.CreatedNode+=k;
        rhs_t_k_table_primarykey.CreatedEdge+=t_k
}
```

Figure 5: (a) *ClassToTable* with VMTS QVT plug-in, (b) Generated VMTS rewriting rule *ClassToTable*, (c) *When* and *Where* clauses of the rule *ClassToTable*, and (d) Generated Imperative OCL code

mations are only possible if an inverse operational implementation is provided separately.

In an inverse transformation executed in VMTS, the modification, creation, and deletion of the model structures can be achieved with swapping LHS and RHS graphs and inverting the modifications described by internal causalities. But the attribute modification is hard in case of QVT as well. In general, the generated target model does not contain all information covered by the source model. Therefore, it is not possible to correctly produce the source model from the target model with an inverse transformation. In summary, general inverse transformation should contain totally new internal causalities that perform attribute modifications.

Bi-directional transformations form the basis of real roundtrip engineering. Forward engineering is the basic model-to-code transformation and reverse engineering is the inverse transformation (code-to-model). Bi-directional transformations support incremental software development, where the changes achieved either in the models or in the code should be synchronized with the other artifacts.

5 Transforming QVT Constructs to VMTS

The transformation that generates VMTS Visual Model Processor from QVT transformation specification is illustrated on the ”class model to relational database management system (RDBMS) model” transformation (also referred to as object-relational mapping). The complete QVT solution for the basic version of the model transformation example can be found in [Que]. The current section introduces the methods that we use to process QVT relations and examples are provided based on the *ClassToRDBMS* transformation.

In Figure 3, the overview of the QVT realization with VMTS is depicted. We use VMTS Presentation Framework (VPF) [VMT] as a basis for the QVT visualization. VPF provides a flexible plug-in-based architecture and offers individual metamodel-dependent visualization and editing features. This facilitates to edit QVT transformations visually in VMTS environment using VMTS QVT plug-in. VMTS Traversing Processor (TP) makes it possible to process the visual QVT transformation specification and produce its textual syntax. Finally, the VMTS QVT compiler is used to generate a VMTS Visual Model Processor (VMP), including the transformation rules and the control flow model from a QVT transformation specification.

Based on the *when* and *where* clauses of the QVT relations, a Control Flow (CF) graph is built to support the creation of the VMTS control flow model (VCFL model). Figure 4a depicts the CF graph created from the relations of the QVT transformation *ClassToRDBMS*. The CF graph describes that model processing should start with the relation *PackageToSchema*, the secondly executed relation must be the relation *ClassToTable*, furthermore, the relations *AttributeToColumn* - *ComplexAttributeToColumn* and *AttributeToColumn* - *SuperAttributeToColumn* can be executed in loop.

QVT transformations differentiate top-level and non-top-level relations. The execution of a transformation requires all its top-level relations to hold, whereas non-top-level relations are required to hold only when they are invoked from the *where* clause of another relation. In the transformation *ClassToRDBMS*, relations *PackageToSchema*, *ClassToTable* and *AssocToFKey* are top-level relations. In Figure 4b, the generated VMTS VCFL model is depicted, where we utilized the hierarchical rule construct of the VMTS approach.

Figure 5a introduces the graphical version of QVT relation *ClassToTable* edited with VMTS QVT plug-in. In VMTS QVT plug-in, *when* and *where* clauses are properties of the transformation rules, they are not shown below LHS and RHS of the rule, because it is not required by the QVT specification, they can be edited separately via a property control (Figure 5c).

The VMTS transformation rules are created from the QVT relations by the VMTS QVT compiler. The inputs are the VMTS QVT plug-in-based relations, and the outputs are the VMTS metamodel-based model transformation rules with a control flow model. Both model elements created with QVT plug-in and the VMTS transformation rule elements refer to the appropriate *UML Class* (*Package*, *Class*, *Attribute*, *PrimitiveDataType*, and *Association*) and *RDBMS* metamodel elements (*Schema*, *Table*, *Column*, *Key*, and *ForeignKey*).

The main steps achieved by VMTS QVT compiler are as follows: (i) the compiler processes the QVT graphical representation and generates QVT textual representation using the Relations language. (ii) From the textual representation abstract syntax tree (AST) is generated, from which the control flow (CF) graph is created. (iii) With the help of the AST and CF graph the compiler generates the VCFL model. (iv) Using the AST the compiler produces VMTS rewriting rules. The generated rules contains LHS and RHS graphs with metatype information (Figure 5b), internal causalities with Imperative OCL codes (Figure 5d), and constraints.

Figure 5b presents the VMTS rewriting rule generated from the QVT relation *ClassToTable*. The rule expresses the same functionality as its corresponding QVT relation. The generated rule contains one internal causality, which, using imperative OCL, describes the operation that should be achieved during the rule execution (Figure 5d). Furthermore, the following OCL constraint is generated to require that LHS match only persistent classes:

```
context Class inv Class_const1:
kind = 'Persistent'
```

For more details please refer to [VMT].

6 Conclusions

This paper has demonstrated the realization of the OMG QVT with metamodel-based model transformation constructs. Not all, but the main QVT constructs have been analyzed from the point of metamodel-based model transformation, and the appropriate VMTS constructs have been introduced that provide the same functionality. The relations between QVT and VMTS constructs have been discussed. Furthermore, the method that we use to create VMTS transformation rules from QVT Relations and VMTS control flow (VCFL) model based on the QVT *when* and *where* clauses have been presented.

Unfortunately, QVT does not have a visual control flow support. Moreover, the branching mechanism provided by the *when-where* clauses is a bit difficult to use. Often model-to-model and model-to-code transformations need to follow an algorithm that requires a stricter control over the execution sequence of the rules, therefore, VCFL models provide a more comfortable way to build transformations from individual rules.

Several open questions remain related to the QVT that should be examined and discussed, for example, the attribute modification in bi-directional or in-place transformations. In general, the inverse transformations cannot be automatically created from the original transformations. Furthermore, a transformation may be considered in-place when its source and target models are both bound to the same model at runtime. In this case the trace objects connect the actual model element with itself, and this means that the original attribute values are lost, since they cannot be queried with the help of the trace objects.

The transformation *QVTConstructToVMTSConstruct* is a QVT compiler implemented in VMTS. We have placed emphasis on the processing of the QVT constructs of the presented approach. The VMTS QVT plug-in can be used to define QVT relations with the necessary attributes and *when* and *where* clauses [VMT].

In summary, we can conclude that QVT can be realized by graph rewriting-based model transformation; furthermore, transformations can be validated with VMTS high-level constructs. Therefore, QVT can utilize the results originating from metamodel-based model transformation and the formal background of graph rewriting.

7 Acknowledgement

The paper is established by the support of the National Office for Research and Technology (Hungary).

References

[dLVA04] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Journal of Software and Systems Modeling*, 3(3):194–209, May 2004.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jorg Kreowski, and Grzegorz Rozenberg. *Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools*, volume 2. World Scientific, Singapore, 1999.

[KASS03] Gábor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. 9(11):1296–1321, 2003.

[KNNZ00] Hans J. Köhler, Ulrich Nickel, Jürg Niere, and Albert Zündorf. Integrating UML diagrams for production control systems. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 241–251, New York, NY, USA, 2000. ACM Press.

[LLC05] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Constraint Validation Support in Visual Model Transformation Systems. *Acta Cybernetica*, 17(2):339–357, 2005.

[MOF] MOF Meta Object Facility. `www.omg.org`.

[OMDA] MDA OMG Model-Driven Architecture. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[OOCL] OCL OMG Object Constraint Language. `http://www.omg.org/docs/ptc/03-10-14.pdf`.

[Que] Transformation Query, View. `http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf`

[Roz97] Grzegorz Rozenberg. *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, Singapore, 1997.

[RS97] Jan Rekers and Andy Schürr. Defining and Parsing Visual Languages with Layered Graph Grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[Tae03] Gabriele Taentzer. AGG: A Graph Transformation Environment for System Modeling and Validation. In *Proc. Tool Exihibition at Formal Methods 2003*, September 2003.

[VMT] VMTS. `http://www.vmts.aut.bme.hu/`.

[VP03] Dániel Varró and András Pataricza. VPM: Mathematics of Metamodeling is Metamodeling Mathematics. *Journal of Software and Systems Modelling*, (1):1–24, 2003. In press.