Model-Based Development with Validated Model Transformation

László Lengyel, Tihamér Levendovszky, Gergely Mezei and Hassan Charaf

Budapest University of Technology and Economics,
Goldmann György tér 3.
1111 Budapest, Hungary

Abstract. Model-Driven Architecture (MDA) as a model-based approach to software development facilitates the synthesis of application programs from models created using customized, domain-specific model processors. MDA model compilers can be realized by graph rewriting-based model transformation. In Visual Modeling and Transformation System (VMTS), metamodel-based transformation steps enables assigning OCL constraints to model transformation steps. Based on this facility, the paper proposes a novel validated model transformation approach that can ensure to validate not only the individual transformation steps, but the whole transformations as well. The discussed approach provides a visual control flow language to define transformations visually in a simple way that results more efficient development process. The presented methods are illustrated using a case study from the field of model-based development.

1 Introduction

Model-driven development approaches (e.g. Model-Integrated Computing (MIC) [1] and OMG's Model-Driven Architecture (MDA) [2] emphasize the use of models at all stages of system development. They have placed model-based approaches to software development into focus.

MIC advocates the use of domain-specific concepts to represent the system design. Domain-specific models are then used to synthesize executable systems, perform analysis or drive simulations. Using domain concepts to represent the system design helps increase productivity, makes systems easier to maintain, and shortens the development cycle.

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 UML and other software models containing enough specification to generate the platform-dependent artifacts automatically by model compilers.

Lengyel L., Levendovszky T., Mezei G. and Charaf H. (2006).
Model-Based Development with Validated Model Transformation.
In *Proceedings of the 2nd International Workshop on Model-Driven Enterprise Information Systems*, pages 39-48
DOI: 10.5220/0002474400390048

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 can conclude that transformations in general play an essential role in model-based development, thus, there is a need for highly reusable model transformation tools. These tools must make the model transformation flexible and expressive, therefore, it should preferably be defined visually. Furthermore they should support control flow, constraints, parameter passing between sequential rules, and conditional branching. Moreover, they should be user friendly and simple to use to make the development as efficient as it is possible.

The approach presented here uses graph rewriting-based visual model transformation. To define the transformation steps precisely and support the validated model transformation beyond the structure of the visual models, additional constraints must be specified which ensure the correctness of the attributes, or other properties can be enforced. Using Object Constraint Language (OCL) [3] constraints provides a solution for these issues. The use of OCL as a constraint and query language in modeling is found to be simple and powerful. We have shown that it can be applied to model transformations as well [4].

The main contribution of the current paper is the validated online model transformation. Section 2 presents the motivation on a real word case study. Section 3 introduces the principles of the validated model transformation: the relation between the pre- and postconditions and OCL constraints propagated to model transformation steps. Section 3.1 shortly presents the Visual Control Flow Language (VCFL) of the Visual Modeling and Transformation System (VMTS) [5] that facilitates an efficient and simple way to define model transformations visually. Using the motivation case study, Section 3.2 discusses the details of the validated model transformation. The approach presented here makes possible to require transformation steps as well as the whole transformations to validate, preserve or guarantee certain properties during the transformation. Section 4 summarizes the related work and compares VMTS with other model transformation approaches. Finally, conclusions are provided.

2 Motivation – A Case Study

To illustrate the motivations on a real word example a case study is provided. The case study is a variation of the “class model to relational database management system (RDBMS) model” transformation (also referred to as object-relational mapping).

The requirements stated against the transformation that it should guarantee are the following properties:

- Classes that are marked as non-abstract in the source model should be transformed into a single table of the same name in the target model. The resultant table should contain one added primary key column, one column for each at-

tribute in the class, and one or more columns for associations based on the next rule.

- In general, an association may, or may not, map to a table. It depends on the type and multiplicity of the association.
  - Many-to-many (N:N) associations, should be mapped to distinct tables. The primary keys for both related classes should become attributes of the association table (foreign keys). Foreign keys do not allow NULL values
  - One-to-many (1:N) and associations, using one or more foreign key columns should be merged into the table for the class on the "many" side.
  - For one-to-one (1:1) associations, also the foreign key should be buried optionally in one of the affected tables.
- Parent class attributes should be mapped into tables created from inherited classes.

The required rules jointly guarantee that the generated database is in third normal form [6].

At the implementation level, system validation can be achieved by testing. Various tools and methodologies have been developed to assist in testing the implementation of a system (for example, unit testing, mutation testing, and white/black box testing). However, in case of model transformation environments, it is not enough to validate that the transformation engine itself works as it is expected. The transformation specification should also be validated.

There are only few and not complete facilities provided for testing offline transformation specifications in an executable style. Related to the expected output there is nothing that can be guaranteed by these transformations. The transformation should be tested: not only the syntactical but the semantical correctness is also required. In fact, the testing requires huge efforts, and even after the testing it is not guaranteed that the transformation produces the expected output for all valid input. The reason is that there is no real possibility that the testing covers all the possible cases. But, in the case of the case study the following issues should be guaranteed by the transformation: (i) Each table has primary key, (ii) each class attribute is part of a table, (iii) each parent class attribute is part of a table created for its inherited class, (iv) each many-to-many association has a distinct table, (v) each one-to-many and one-to-one association has merged into the appropriate tables, (vi) foreign keys not allow NULL value, and (vii) each association class attribute buried into the appropriate table based on the multiplicities of its association.

There is a need for a solution that can validate model transformation specifications: online validated model transformation that guarantees if the transformation finishes successfully, the generated output (database schema) is valid, and it is in accordance with the requirements above.

3 Validated Model Transformation

Graph rewriting [7] 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 the rule being applied to (host graph), and replacing this subgraph with RHS.

The Object Constraint Language is a formal language for the analysis and design of software systems. It is a subset of the UML standard [8], and OCL allows software developers to write constraints and queries over object models. A precondition to an operation is a restriction that must be true immediately prior to its execution. Similarly, a postcondition to an operation is a restriction that must be true immediately after its execution.

A *precondition* assigned to a transformation step is a *boolean* expression that must be true at the moment when the transformation step is fired. Similarly, a *postcondition* assigned to a transformation step is a *boolean* expression that must be true after the completion of a transformation step. If a *precondition* of a transformation step is not true then the transformation step fails without being fired. If a *postcondition* of a transformation step is not true after the execution of the transformation step then the transformation step fails. A direct corollary of this is that an OCL expression in LHS is a precondition to the transformation step, and an OCL expression in RHS is a postcondition to the transformation step. A transformation step can be fired if and only if all conditions enlisted in LHS are true. Also, if a transformation step finished successfully then all conditions enlisted in RHS must be true [4].

3.1 VMTS Visual Control Flow Language

VMTS is an n-layer metamodeling environment which supports editing models according to their metamodels, and allows specifying 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 step during the model transformation process.

Model-to-model transformations often need to follow an algorithm that requires a stricter control over the execution sequence of the steps. The VMTS approach is a visual approach and it also uses graphical notation for control flow: stereotyped UML activity diagram [8]. VMTS Visual Control Flow Language (VCFL) is a visual language for controlled graph rewriting and transformation, which supports the following constructs: sequencing transformation steps, branching with OCL constraints, hierarchical steps, parallel execution of the steps, and iteration.

The branching construct is required, because often, the transformation that we would like to apply depends on a condition. In VCFL, OCL constraints assigned to the decision elements can choose between the paths of optional numbers, based on the properties of the actual host model and the success of the last transformation step (*SystemLastRuleSucceed*).

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

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 step. 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.

The interface of the transformation steps allows the output of one step to be the input of another step (parameter passing). In VCFL, this construction is referred to as *external causality*. This feature accelerates the matching and reduces the complexity.

3.2 Validated Solution of the Case Study

In this section, a validated solution for the transformation *Class2RDBMS* supported by VCFL is presented. The case study follows the entity-driven database design and the existence-based identity implementation [6]. The metamodel for class models is shown in Fig 1a. A model consists of classes and relations between them (*Inheritance*, *Association* and *Dependency*). The *MetaClass* attributes describes the following. A class can be abstract, and it consists of *ClassAttributes* and *ClassOperations*.

The metamodel for RDBMS models is depicted in Fig. 1b. An RDBMS model consists of one or more tables. A table consists of one or more columns, which are defined as attributes of the metatype *Table*.

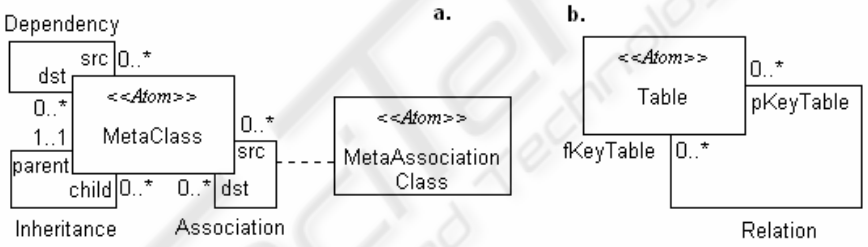

Fig. 1. VMTS Class diagram and Relational Database metamodels.

An example input and its required output model are depicted in Fig. 2. In the input model, the classes *Inhabitant* and *Institute* are abstract. The relation between the classes *Adult* and *Institute* is N:N. In Fig. 2b, there is a table for each non-abstract class and there are two connection tables for the N:N relationships (tables *Adult_School* and *Adult_Company*). Each table enlists its columns and their data type.

The control flow model of the case study (Fig. 3) can be divided into three parts according to the goal of the units. (i) The large loop on the top is responsible for the table creation and inheritance-related issues. (ii) The step *ProcessAssociation* processes the associations. (iii) Finally, the last steps remove the helper nodes and temporary associations.

One of the major challenges is to process the inheritance hierarchy properly, so the transformation must traverse the inheritance chains, because the parent class association should be taken into account recursively by subclasses.

The first step (*CreateTable*) is depicted in Fig. 4a. It matches a non-abstract class and creates a table based on it.

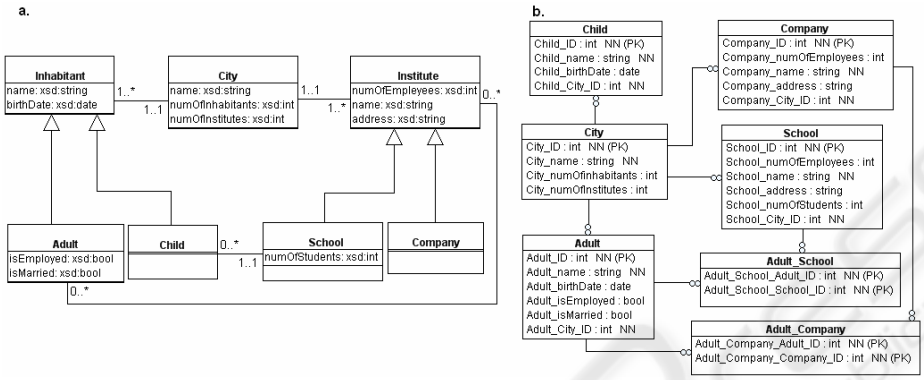

Fig. 2. (a) Example input of the case study, (b) Required output of the example input model.

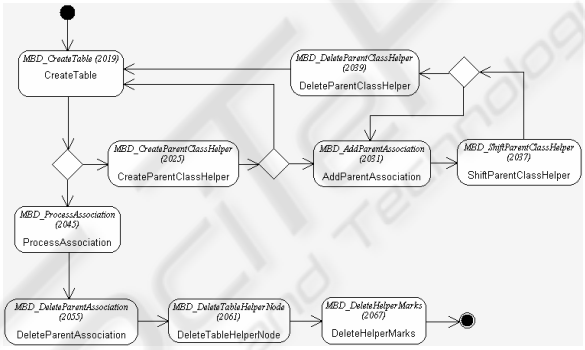

Fig. 3. The VCFL model of the transformation *Class2RDBMS*.

To require certain properties of the transformation step *CreateTable* the following constraints are applied:

```
context Class inv NonAbstract:
not self.abstract
```

The constraint *NonAbstract* is assigned to the pattern rule node (PRN) *Class* in LHS of the step *CreateTable*. This link forms a precondition, it requires the step to process only non-abstract classes.

```
context Table inv PrimaryKey:
self.columns->exists(c | c.datatype = 'int' and c.is_primary_key)
```

The constraint *PrimaryKey* is a postcondition of the step *CreateTable*, it is assigned to the PRN *Table*. This *guarantee* type constraint requires the step that all created table has a primary key of *int* type.

```
context Table inv PrimaryAndForeignKey:
not self.columns->exists(c | (c.is_primary_key or
c.is_foreign_key) and c.allows_null)
```

The constraint *PrimaryAndForeignKey* of *guarantee* type is also a postcondition that necessitates the primary and foreign key columns do not allow NULL values.

```
context Atom inv ClassAttrsAndTableCols:
self.class.attribute->forAll(self.table.column->
exists(c | (c.columnName = class.attribute.name))
```

The *guarantee* type constraint *ClassAttrsAndTableCols* is linked to the PRN *TableHelperNode*, it requires that each class attribute should have a created column with the same name in the resultant table.

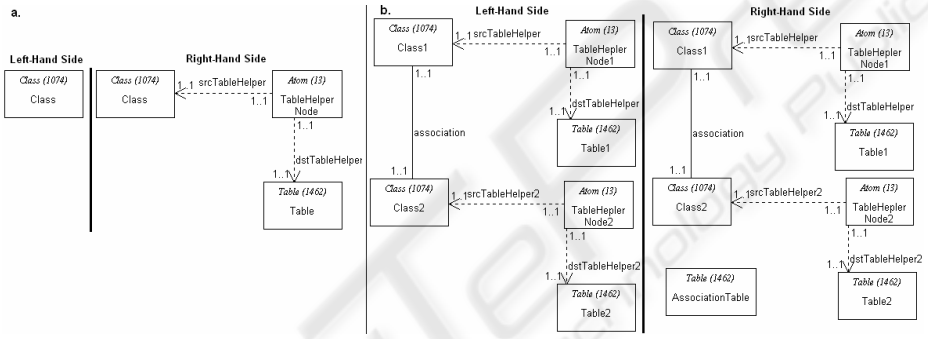

Fig. 4. Transformation steps (a) *CreateTable* and (b) *ProcessAssociation*.

If the step *CreateTable* was successful, the decision object selects the branch pointing to the step *CreateParentClassHelper*, otherwise it selects *ProcessAssociation*.

Step *AddParentAssociation* creates a temporary association that links the subclass to the neighbors of the parent class. These associations facilitate that the step *ProcessAssociations* processes not only the direct associations of a class, but the association of its parents as well.

The external causalities defined between the steps *ShiftParentClassHelper* and *AddParentAssociation* are depicted in Fig. 5. The *ParentClassHelperNode* connects a subclass with its parent class, but the parent class can also have a parent. The transformation must traverse the whole inheritance hierarchy. The step *ShiftParentClassHelper* removes the original *ParentClassHelperNode* and adds a new one which links the subclass to the parent of the parent class.

The step *ProcessAssociation* (Fig. 4b) uniformly processes the associations and the helper parent associations as well. It creates association tables (N:N associations), and

completes the already existing tables (1:N and 1:1 associations) with new foreign key columns. The following constraints are assigned to the step *ProcessAssociation*:

```
context Association inv OneToOneOrOneToMany:
(self.leftMaxMultiplicity = '1' or self.rightMaxMultiplicity = '1')
implies self.attribute->forAll (self.class1.helperNode.table.column->
exists(c | (c.columnName = attribute.name)) or self.attribute->forAll
(self.class2.helperNode.table.column->exists(c | (c.columnName = at-
tribute.name))
```

The constraint *OneToOneOrOneToMany* guarantees that the attributes of the one-to-one and the one-to-many association are buried into one of the tables created for the classes connected by the actually processed association.

```
context Association inv ManyToMany:
(self.leftMaxMultiplicity = '*' and self.rightMaxMultiplicity = '*')
implies self.attribute->forAll(self.class1.helperNode.table. con-
nectTable.column->exists(c | (c.columnName = attribute.name))
```

The constraint *ManyToMany* guarantees that, for each many-to-many type association in the resulted model, there is a distinct table. Furthermore, the table contains all attributes of the association with the same name.

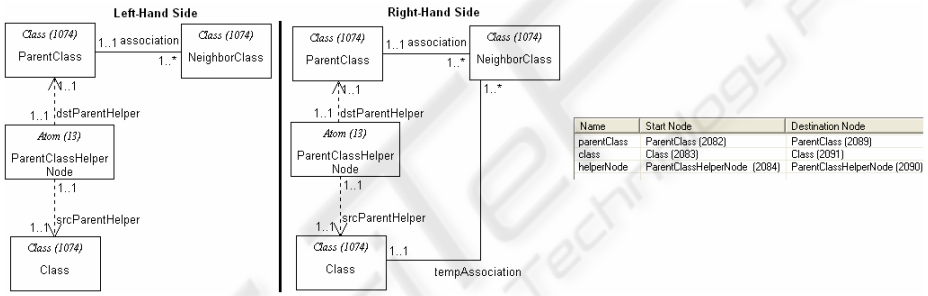

Fig. 5. Transformation step *AddParentAssociation* and external causalities between steps *ShiftParentClassHelper* and *AddParentAssociation*.

The last three transformation steps remove the remaining instances of the helper nodes, and restore the original properties of the class model elements. As a result of these steps, the input model becomes free of any helper structure.

The constraints assigned to the transformation steps guarantee the requirements from Section 2. As it is presented, after a successful step execution the conditions hold and the output is valid that cannot be achieved without constraints.

4 Related Work and Comparison

Many approaches have been introduced in the field of graph grammars and transformations to capture graph domains; for instance, the GReAT [9], the PROGRES [10],

the FUJABA [11], the VIATRA [12], and AGG [13]. These approaches are specific to the particular system, and each of them has some features that others do not offer.

The GReAT framework is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts. The control structure of GReAT allows specifying an initial context for matching to reduce the complexity of the general matching case. PROGRES is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. In FUJABA, a combination of activity diagrams and collaboration diagrams (story-diagrams) are used to express control structures. VIATRA is a model transformation framework, its attribute transformation is performed by abstract state machine statements, and there is built-in support for attributes of basic Java types. AGG is a visual tool environment consisting of editors, interpreter and debugger for attributed graph transformation; attribute computation by Java. The control structure of AGG is given by layers.

The Model-Driven Architecture offers a standard interface to implement model transformation tools. The transformation related part of MDA is the Query, Views, Transformation for MOF 2.0 [14]. Three types of operations are provided: *queries* on models, *views* on metamodels and *transformation* on models.

Compared to other approaches, VMTS meets the expectations in model-to-model and model-to-code transformation. VMTS has state of the art mechanisms for validated model transformation, constraint management and control flow definition. It has several standalone algorithms and other solutions that make them efficient.

VMTS has a unique constraint management and online transformation validation support. It provides a high-level control flow language with several constructs that optimize and make the transformations highly configurable: external causalities, efficient branch selecting, and pivot nodes. 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 step 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, there is no need to define test rules.

5 Conclusions

Model-based development necessitates the transformation of models between different stages of the design process. These transformations must be precisely – preferably visually – specified. In this paper, a graph-transformation-based technique for specifying such a model transformation is presented. It has been shown that VMTS provides a high level visual language to define transformations in an easy way. In the provided control flow approach the transformations are represented in the form of explicitly sequenced transformation steps. We have shown the fundamental concepts of the VMTS approach, namely, the metamodel-based model transformation steps, the external- and internal-causalities for parameter passing, constraint support, and conditional branching with OCL constraints.

The main result of the paper is illustrating online validated model transformation that applying OCL constraints propagated to transformation steps facilitates to require the whole transformations to validate, preserve or guarantee certain model properties.

VCFL has already been applied in MDA-based industrial projects successfully, such as generating user interface from resource model, user interface handler code from statechart model for Symbian [15], and .NET CF mobile platforms [4].

Acknowledgements

The activities described in this paper supported, in part, by Information Technology Innovation and Knowledge Centre.

References

1. J. Sztipanovits, and G. Karsai, Model-Integrated Computing, IEEE Computer, Apr. 1997, pp. 110-112.
2. OMG MDA Guide Version 1.0.1, OMG, doc. number: omg/2003-06-01, 12th June 2003, http://www.omg.org/docs/omg/03-06-01.pdf
3. OMG Object Constraint Language Spec. (OCL), www.omg.org
4. L. Lengyel, T. Levendovszky, H. Charaf, Implementing an OCL Compiler for .NET, In Proceedings of the 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005, pp. 121-130.
5. The VMTS Homepage. http://avalon.aut.bme.hu/~tihamer/research/vmts
6. Michael R Blaha, and William Premerlani, Object-Oriented Modeling and Design for Database Applications, Prentice Hall, 1998.
7. G. Rozenberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol.1 World Scientific, Singapore, 1997.
8. OMG UML 2.0 Specifications, http://www.omg.org/uml/
9. G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, On the Use of Graph Transformation in the Formal Specification of Model Interpreters, Journal of Universal Computer Science, 2003.
10. J. Reekers, A. Schürr, Defining and Parsing Visual Languages, Journal of Visual Languages and Computing, 8, Academic Press, 1997, pp. 27-55.
11. H. J. Köhler, U. A. Nickel, J. Niere, A. Zündorf, Integrating UML Diagrams for Production Control Systems, ICSE, Limerick Ireland, ACM Press, 2000, pp. 241-251.
12. D. Varró and A. Pataricza, "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML", SoSyM, 2003.
13. G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, In J. Pfaltz, M. Nagl, and B. Boehlen (eds.), Application of Graph Transformations with Industrial Relevance (AGTIVE'03), vol. 3062. Springer LNCS, 2004.
14. OMG Query/View/ Transformation. http://www.omg.org/docs/ptc/05-11-01.pdf.
15. L. Lengyel, T. Levendovszky, G. Mezei, B. Forstner, H. Charaf, Metamodel-Based Model Transformation with Aspect-Oriented Constraints, International Workshop on Graph and Model Transformation, GraMoT, ENTCS Vol. 152, Tallinn, Estonia, 2005, pp. 111-123.