Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Test-driven verification/validation of model transformations*

László LENGYEL‡, Hassan CHARAF

(*Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest 1117, Hungary*)

E-mail: lengyel@aut.bme.hu; hassan@aut.bme.hu

Received Mar. 26, 2014; Revision accepted Dec. 15, 2014; Crosschecked Dec. 30, 2014

Abstract: Why is it important to verify/validate model transformations? The motivation is to improve the quality of the transformations, and therefore the quality of the generated software artifacts. Verified/validated model transformations make it possible to ensure certain properties of the generated software artifacts. In this way, verification/validation methods can guarantee different requirements stated by the actual domain against the generated/modified/optimized software products. For example, a verified/validated model transformation can ensure the preservation of certain properties during the model-to-model transformation. This paper emphasizes the necessity of methods that make model transformation verified/validated, discusses the different scenarios of model transformation verification and validation, and introduces the principles of a novel test-driven method for verifying/validating model transformations. We provide a solution that makes it possible to automatically generate test input models for model transformations. Furthermore, we collect and discuss the actual open issues in the field of verification/validation of model transformations.

Key words: Graph rewriting based model transformations, Verification/validation, Test-driven verification
doi:10.1631/FITEE.1400111 **Document code:** A **CLC number:** TP311

1 Introduction

We often project a model into another domain or format, for example, into a formal model. And we always ask, what ensures that the projection is free of conceptual errors? The central question of the area is the following: how can we ensure that the model transformation does what it is intended to do? This paper is intended to contribute to answering this question.

Model transformations have a wide application field. Different transformation solutions support one or more from the following scenarios: refining the design to implementation (OMG, 2014), transforming models into other domains (Varró and Pataricza, 2003), aspect weaving (Assmann and Ludwig, 2000), analysis and verification (Assmann, 1996), software refactoring (van Gorp *et al.*, 2003), simulation and execution of a model, querying some information and providing a view for it, abstracting models, assigning concrete representation to model elements, migration, normalization, optimization, and synchronization (Amrani *et al.*, 2012).

As model transformations are being applied to so many different scenarios, there is a compelling need for methods regarding their development, and also for verifying/validating them. Verification and validation of a model transformation is the process of ensuring that the transformation definition meets requirements and fulfills its defined role.

The goals of the transformations' analysis are to show that, in the case of a valid input model, certain properties will be true for the output model. The analysis of a transformation is said to be static when the implementation of the transformation and the language definition of the input and output models are used during the analysis process, but we do not take concrete input models into account. In the case of a

‡ Corresponding author

* Project partially supported by the European Union and the European Social Fund (No. TAMOP-4.2.2.C-11/1/KONV-2012-0013)

ORCID: László LENGYEL, http://orcid.org/0000-0002-5129-4198; Hassan CHARAF, http://orcid.org/0000-0002-8911-0219

dynamic approach, we analyze the transformation for a specific input model, and then check whether certain properties hold for the output model during or after the successful application of the transformation. The static technique is more general and poses more complex challenges. The goal of static analysis is to determine if the transformation itself meets various, specific requirements.

In model-driven system development, a software design and analysis process involves designing the system, projecting it into the analysis domain, and executing the verification on the analysis model. Model transformation is a powerful and convenient method frequently used to automate this conversion (Narayanan and Karsai, 2008). However, the verification of different properties at the model level is useful only if automatic code generation is guaranteed to be correct. This means that the verified properties should be true for the generated code as well (Giese *et al.*, 2006).

For most computer-controlled systems, an effective design process requires an early validation of the concepts and architectural choice. However, a standard modeling language alone does not guarantee the correctness of the design. Therefore, during the design of software systems, the design models are frequently projected into various mathematical domains to perform formal analysis of the system under design via automatic model transformations (Varró and Pataricza, 2003).

To summarize, it is significant to understand that model transformations themselves can be incorrect; therefore, uncovering solutions to make model transformations free of conceptual errors is crucial.

2 Background

This section introduces the classification of model transformation approaches, discusses the basics of graph rewriting based model transformation, and summarizes the ideas and motivations related to testing model transformations.

2.1 Classification of model transformation approaches

There are several model transformation approaches ranging from relational specifications (Akehurst and Kent, 2002) to graph transformation techniques (Ehrig *et al.*, 1999), and to algorithmic techniques for implementing a model transformation. Based on Czarnecki and Helsen (2006) and Mens and van Gorp (2006), we distinguish between the following approaches (Fig. 1):

Traversal-based and direct manipulation approaches: These model processing approaches provide mechanisms to visit the internal representation of a model and write text (source code or other text, e.g., XML) to a stream while optimizing and generating models and other artifacts. Furthermore, modeling and model processing approaches (aside from the model representation) offer some application programming interfaces (APIs) to manipulate the models. These approaches are usually implemented using an imperative programming language (Vajk *et al.*, 2009).

Template-based approaches: Approaches in this category are applied mainly in the case of model-to-code generation. A template usually consists of the target text containing splices of source code (meta-code) used to access information from the source and to perform code selection and iterative expansion. The meta-code may be imperative program code or declarative queries as is the case with OCL (OMG, 2012), XPath, or T4 text templates.

Relational approaches: These approaches declaratively map between source and target models. This mapping is specified by constraints, which define the expected results, not the way in which they are achieved. Some examples of this are Query, Views, Transformations (QVT) (OMG, 2011) and partially Triple Graph Grammars (TGGs) (Schürr, 1994).

Graph rewriting based approaches: Models are represented as typed, attributed, labeled graphs. The theory of graph transformation is used to transform models. Some examples of these approaches are AGG

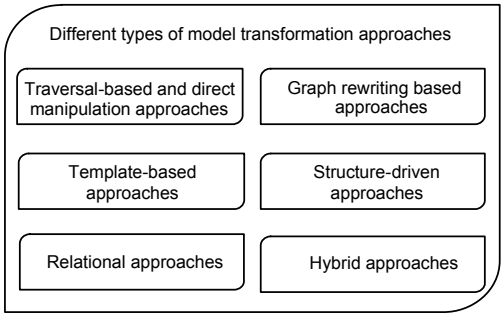

Fig. 1 Classification of model transformation approaches

(http://www.user.tu-berlin.de/o.runge/AGG/), AToM^3 (http://atom3.cs.mcgill.ca), GReAT (http://www.isis.vanderbilt.edu/tools/GReAT), TGGs, VIATRA2 (http://eclipse.org/gmt/VIATRA2), and VMTS (http://www.aut.bme.hu/vmts).

Structure-driven approaches: The transformation is performed in phases: the first phase is concerned with creating the hierarchical structure of the target model, whereas the second phase sets the attributes and references for the target, e.g., OptimalJ and QVT (OMG, 2011).

Hybrid approaches: Hybrid approaches combine two or more of the previous categories. For example, ATL (http://eclipse.org/atl/) combines template-based, direct manipulation, and graph rewriting based approaches. Another hybrid approach worth mentioning is TGGs.

One of the most popular model transformation approaches, taking both the literature and the industry into consideration, is the graph rewriting based approach. In this method, the concentration is on the verification and validation capabilities of the graph transformation based approaches. Therefore, the next section summarizes the theoretical foundations of this approach.

2.2 Graph rewriting based model transformation

Graph rewriting based transformation is a widely used technique for model transformation (Karsai *et al.*, 2003; de Lara *et al.*, 2004). Graph transformation has its roots in classical approaches to rewriting, such as Chomsky grammars and term rewriting (Rozenberg, 1997). There are many other representations of this, which are not yet mentioned. In essence, a rewriting rule is composed of a left-hand side (LHS) pattern and a right-hand side (RHS) pattern.

Operationally, a graph transformation from a graph G to a graph H follows these main steps: (1) Choose a rewriting rule; (2) Find an occurrence of the LHS in G satisfying the application conditions of the rule; (3) Replace the subgraph matched in G by RHS.

There are many different graph transformation approaches applying the above steps (Rozenberg, 1997; Syriani, 2009). One of them is the popular algebraic approach, based on category theory with push-out constructs on the category (Ehrig *et al.*, 2006). Algebraic graph transformations have two branches, i.e., single-push-out (SPO) and double-push-out (DPO) approaches.

The DPO approach has a large variety of graph types and other kinds of high-level structures, such as labeled graphs, typed graphs, hypergraphs, attributed graphs, Petri nets, and algebraic specifications. This extension from graphs to high-level structures was initiated in Ehrig *et al.* (1991a; 1991b), leading to the theory of high-level replacement (HLR) systems. In Ehrig *et al.* (2004), the concept of HLR systems was joined with adhesive categories, introduced by Lack and Sobocinski (2004), leading to the algebraic construct of adhesive HLR categories and systems. In general, an adhesive HLR system is based on the DPO method. However, these are not only for the category of graphs (also called rules), which describe abstractly how objects in this system can be transformed. Ehrig *et al*. (2006) provided a detailed presentation of adhesive HLR systems. In the context of this paper, it is relevant only for typed, attributed graphs.

Graph transformations define the transformation of models. The LHS of a rule defines the pattern to be found in the host model; therefore, the LHS is considered the positive application condition (PAC). However, it is often necessary to specify what pattern should not be present. This is referred to as the negative application condition (NAC) (Habel *et al.*, 1996). Besides NACs, some approaches such as AGG and VIATRA2 use other constraint languages, e.g., OCL, to define the execution conditions.

The scheduling of transformation rules can be achieved by explicit control structures or can be implicit, due to the nature of their rule specifications. Moreover, several rules may be applicable at the same time. Blostein *et al.* (1996) have classified graph transformation organization in four categories: (1) An unordered graph-rewriting system simply consists of a set of graph-rewriting rules. Applicable rules are selected non-deterministically until none are applicable. (2) A graph grammar consists of the rules, a start graph, and terminal states. Graph grammars are used for generating language elements and language recognition. (3) In ordered graph-rewriting systems, a control mechanism explicitly orders the rule application of a set of rewriting rules (e.g., priority-based, layered/phased, or with an explicit control flow structure). (4) In event-driven graph-rewriting systems, rule execution is triggered by external events. This approach has recently seen a rise in popularity (Guerra and de Lara, 2007).

Controlled (or programmed) graph transformations impose a control structure over the transformation rules to maintain a stricter ordering over the execution of a sequence of rules. The control structure primitives of a graph transformation may provide the following properties: atomicity, sequencing, branching, looping, non-determinism, recursion, parallelism, back-tracking, and/or hierarchy (Rozenberg, 1997; Lengyel, 2006).

Some examples of control structures are as follows: AGG uses layered graph grammars. The layers fix the order in which rules are applied. The control mechanism of AToM^3 is a priority-based transformation flow. Fujaba (http://www.fujaba.de/) uses story diagrams to define model transformations. The control structure language of GReAT uses a dataflow diagram notation. GReAT also has a test rule construction; a test rule is a special expression that is used to change the control flow during execution. VIATRA2 applies abstract state machines (ASMs). VMTS uses stereotyped Unified Modeling Language (UML) (OMG, 2010) activity diagrams to further specify control flow structures. The model transformation process is depicted in Fig. 2. In Taentzer *et al.* (2005), a comparative study was provided that examines the control structure capabilities of the tools AGG, AToM^3, VIATRA2, and VMTS.

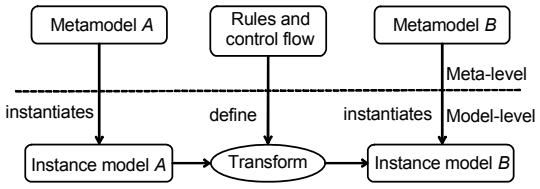

Fig. 2 Model transformation process

2.3 Testing model transformation

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test (Kaner, 2006). Testing can never completely identify all the defects within software (Pan, 1999). Instead, it compares the state and behavior of the artifact by which someone (the software engineer or the domain specialist) might recognize a problem (Leitner *et al.*, 2007).

Testing model transformation is any activity aimed at evaluating a property or behavior of a model processor and determining that it meets its required results. The difficulty in the testing of model transformations stems from the complexity. Testing is more than just debugging the execution of the transformation. The purposes of testing are quality assurance and verification/validation (Hetzel, 1998).

A reasonable part of the defects in transformations is the design error. Bugs on software artifacts, including model transformations, will almost always exist in any software component with acceptable size. This is not because architects and engineers are careless or irresponsible, but because the complexity of software artifacts is generally hard to manage. Humans have only limited ability to handle it. It is also true that for any complex systems, design defects can never be completely eliminated (Kaner, 2006).

Regardless of the limitations, testing is an integral part of model transformation development. In our context testing is usually performed to improve the quality and verify/validate transformations.

Testing is heavily used as a tool in the process of verifying and validating software artifacts. There is no way to directly test quality, but we can test related issues to measure the quality level.

3 Scenarios of model transformation verification and validation

In this section, we discuss the different scenarios of model transformation verification and validation. We refer to these scenarios as 'paths'. Fig. 3 depicts the paths: the top half of the figure represents the operational part, and the bottom half depicts the verification/validation (V&V) part. The operational part is designed by the transformation engineer.

The related verification/validation questions are as follows: Can we verify a property in one of the operational domains (e.g., in source model M_1, transformation T, or target model M_2)? If not, what other domains need to be involved (which path of Fig. 3 should be taken), and where the verification/validation can be performed or more aptly formed? In which way is the mapping arranged between the operational and verification/validation domains?

Sometimes properties that will be verified/validated cannot be expressed in the operational domains. To address this, we have introduced the V&V part including additional domains. In Fig. 3, MM_1 and

MM_2 are the language specifications (metamodels) and can define only two domains. In special cases, MM_1 and MM_2 can be identical. M_1 and M_2 are instance models of MM_1 and MM_2, respectively. The instantiation is defined by the mappings i_1 and i_2. Transformation T converts M_1 into M_2. The mapping trace stores the relationship between elements of models M_1 and M_2. Based on this mapping, for each element of model M_1, we can identify the appropriate target model elements (image) in model M_2, and vice versa; for each element of model M_2, we can identify the appropriate source model elements in model M_1. Our goal is to verify the semantic correctness of transformation T; therefore, if the formalism used by M_1, M_2, and T is not adequate, then they are mapped into a different semantic domain. Their images are S_1, S_2, and TS, respectively. The mapping is defined by ms_1, ms_2, and tts. The correspondence between S_1 and S_2 is a specific knowledge: a special semantic relationship expected by the transformation designer. This correspondence is verified in a semantic domain. Next, assuming that the mappings (ms_1, ms_2, and tts) from the domain-specific artifacts (M_1, M_2, and T) into the semantic domains (S_1, S_2, and TS) are correct, we can reason the correctness of transformation T.

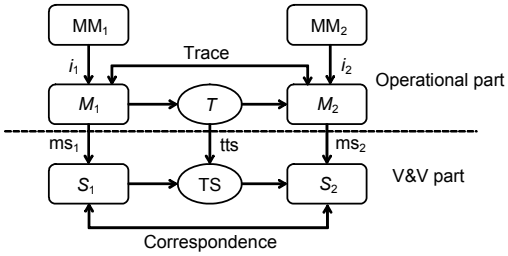

Fig. 3 The paths of model transformations

Recall that in Fig. 3 we have demonstrated a general case scheme, incorporating several special cases. The introduced scheme represents a one-way transformation, but the bi-directional scenario can be constructed by repeating this structure in the opposite direction. In a general case, the two directions require different mappings; only in special cases can the same mapping be applied.

Based on the architecture of the paths, we have identified the following semantic verification/validation types: (1) verification of the models M_1 and M_2 ($\text{Path}^{\text{Models}}$); (2) verification of the transformation T ($\text{Path}^{\text{TransT}}$); (3) verification of the transformation TS ($\text{Path}^{\text{TransTS}}$); (4) verification of the correspondence ($\text{Path}^{\text{Corresp}}$); (5) hybrid verification, with two or more of the previous verification types being combined ($\text{Path}^{\text{Hybrid}}$).

Each of these verification/validation types defines a path. During the verification and validation, we traverse the paths of the framework in the following ways:

$\text{Path}^{\text{Models}}$: Verification of models M_1 and M_2 means M_1 and T (M_1) are verified separately. This type of verification does not attempt to prove the validity of graph transformation T, but verifies that both of the models provide an appropriate solution to the problem. Typically, the conformance into metamodels is validated with this path: the modeling tool allows the creation of appropriate model elements only, while a constraint checker (e.g., OCL checker) is executed on the source model (M_1). Next, transformation T processes the model and generates the output model (M_2), which conforms to the output metamodel (MM_2). Output model M_2 is validated again in the modeling environment: validation of metamodel convergence, including constraint checking.

$\text{Path}^{\text{TransT}}$: Most of the verification/validation approaches aim to check the correctness of the transformation rules in general. There are both static (offline) and dynamic (online) approaches. For example, Asztalos *et al.* (2010a) developed a formal language that is able to express a set of model transformation properties. Basically, the language is appropriate to specify both the properties of the output models and the properties of the relationship between the input and output model pairs. They introduced a final formula which describes the properties that remain true at the end of the transformation. The approach is able to derive the proof or refutation of a verifiable property from the final formula. An example dynamic approach was provided by Lengyel (2006), in which the validation of the transformation is achieved with constraints assigned to the transformation rules as pre- and postconditions.

$\text{Path}^{\text{TransTS}}$: In the most generic case, transforming models into other domains means a projection from the source language to the target language, possibly with an intentional loss of information. Therefore, in certain cases, proving full semantic equivalence between source and target models is not the

objective. Instead, we can define transformation or language-specific (source and target domain) validation properties that should be satisfied by the transformation.

The transformation definition describes the required model manipulation in either an imperative or a declarative (mostly relational) way. This representation is often inappropriate as a subject of verifying certain properties. Therefore, we map the transformation to a domain more suitable to perform formal verification/validation. There are several approaches that map M_1, M_2, and T into a semantic domain and perform the verification either on the image of the transformation (TS: $\text{Path}^{\text{TransTS}}$) or on the correspondence between the images of the source and generated models (S_1 and S_2: $\text{Path}^{\text{Corresp}}$).

An example of $\text{Path}^{\text{TransTS}}$ was provided in Varró *et al.* (2006) in which model transformations were mapped into Petri nets with the goal of performing termination analysis in a more appropriate domain.

$\text{Path}^{\text{Corresp}}$: The correspondence relationship between S_1 and S_2 is domain-specific knowledge; this is the semantics expected by the language and/or model transformation designer. We should realize that the source and the target domains (MM_1 and MM_2) could be quite distant from each other (e.g., abstraction level, domain concepts, or model structure). Thus, the correspondence may be an optional domain-specific knowledge that represents the semantic mapping between the images of source and target models in a semantic domain.

In the context of our verification/validation classification framework, the equilibrium (property preservation) between the source and generated models, which most of the approaches (e.g., Varró and Pataricza (2003), de Lara and Taentzer (2004), and Giese *et al.* (2006)) attempt to verify, is a special mapping among the source and the target domains. Similarly, other special mappings have already been configured, e.g., bi-similarity: two systems can be said to be bi-similar if they behave in the same way; i.e., one system simulates the other, and vice versa (Narayanan and Karsai, 2008).

An example for $\text{Path}^{\text{Corresp}}$ is the following: within the domains of the source and the target modeling languages, it is hard to prove the correctness of the design. Therefore, the models are projected into a formal domain, such as transition systems, and the formal analysis is performed in this domain, e.g., by applying bi-simulation (Narayanan and Karsai, 2008).

$\text{Path}^{\text{Hybrid}}$: This path combines two or more of the paths introduced above. For example, a certain development scenario requires the verification/validation of both transformation termination and some domain-specific properties. $\text{Path}^{\text{TransTS}}$ is applied to verify termination and $\text{Path}^{\text{TransT}}$ to validate the required domain-specific properties, e.g., attribute value requirements.

4 Dynamic validation method

Model transformation rules can be made more relevant to software engineering models if the transformation specifications allow assigning validation constraints to the transformation rules.

An example rule that assembles database models from UML class diagrams is depicted in Fig. 4. Constraints are assigned to the rules: Cons_C1, Cons_C2, Cons_H1, Cons_T1, and Cons_T2. These constraints require the rule to meet different properties (Asztalos *et al.*, 2010b).

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

The constraint NonAbstract (Cons_C1) is a precondition. It requires the rule 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 (Cons_T1) is a postcondition. This rule requires that all the tables should have a primary key of type int.

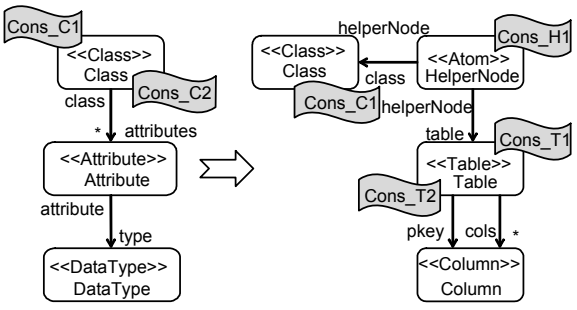

Fig. 4 Example transformation rule: ClassToTable

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

The constraint ClassAttrsAndTableCols (Cons_H1) is propagated to the node TableHelperNode. It requires that each class attribute should have a created column with the same name in the resultant table.

The constraints propagated to the rule guarantee certain properties. After a successful rule execution, the conditions should hold; i.e., the output should be valid. The successful execution of the rule guarantees that the valid output cannot be achieved without these validation constraints.

Dynamic validation covers both the attribute value and the structure validation, which can be expressed in first-order logic extended with traversing capabilities. Example languages currently applied for defining attribute value and interval conditions are Object Constraint Language (OCL), C, Java, and Python. These conditions and requirements are pre- and postconditions of a transformation rule.

Definition 1 (Precondition) A precondition assigned to a rule is a Boolean expression that must be true at the moment of rule firing.

Definition 2 (Postcondition) A postcondition assigned to a rule is a Boolean expression that must be true after the completion of a rule.

If a precondition of a rule is not true, then the rule fails without being fired. If a postcondition of a rule is not true after the execution of the rule, the rule fails.

With pre- and postconditions, the execution of a rule is as follows (Fig. 2): (1) Find the match according to the LHS structure. (2) Validate the constraints defined in LHS on the matched parts of the input model. (3) If a match satisfies all the constraints (preconditions), then execute the rule; otherwise, the rule fails. (4) Validate the constraints defined in RHS on the modified/generated model. If the result of the rule satisfies the postconditions, then the rule is successful; otherwise, the rule fails.

A direct corollary is that an expression in LHS is a precondition to the rule, and an expression in RHS is a postcondition to the rule. A rule can be executed if, and only if, all conditions enlisted in LHS are true. Also, if a rule finishes successfully, then all conditions enlisted in RHS must be true.

This method can be followed in Fig. 4. Finding the structural match the preconditions Cons_C1 and Cons_C2 are validated, and after performing rewriting, postconditions Cons_C1, Cons_H1, Cons_T1, and Cons_T2 are validated. Both of the validations should be successful in order for the whole rule to be successful.

With this method, the required properties can be defined at low level, i.e., on the level of the rules. In summary, we can state that the presented dynamic approach guarantees that if the execution of a rule finishes successfully, the generated output is valid and fulfills the required conditions. The validation of the transformations can be achieved with constraints assigned to the rules as pre- and postconditions.

5 Test-driven validation approaches

The main purpose of testing is to catch software failures (Kaner *et al.*, 1990). The scope of model transformation testing often includes analysis of the transformation definition, execution of that transformation in different conditions. Results derived from testing may also be used to correct the process by which the transformations are developed (Kolawa and Huizinga, 2007).

The goal of the test-driven validation approach is to test graph rewriting based model transformations by automatically generating appropriate input models, executing the transformations, and involving domain specialists to verify the output models based on the input models. It is important that the semantic correctness of the output models cannot be automatically verified; i.e., we need the domain specialists during both the transformation design and testing.

The test-driven validation method needs a model transformation definition and the metamodels of both the input and output domains. The method automatically generates input models that cover all execution paths of the transformation. Covering the whole transformation means that each of the rules in the transformation will be executed at least once. Furthermore, each of the decision points (branching points, forks) is evaluated for both true and false branches; i.e., all of the paths in the control flow model are traversed. The generated input models represent a set of input models. We use the expression

'set' for a bunch of input models that cover the whole transformation. The number of the models in the sets can vary based on the actual domain and also on the actual transformation definition. An objective of the solution is to make these model sets minimal, i.e., to minimize the number and the size of these models.

The method should generate those typical models that effectively cover the whole transformation. Executing the transformation for these input models we obtain the results of the transformation executions. At this point domain specialists are involved. We provide the input model and output model pairs to the domain specialists. Then, based on the input and outputs, and not considering the transformation definition, they can decide whether the transformation does the right processing. Without domain specialists, we cannot verify that the output model is really right, i.e., which is the appropriate output for a given input model.

As we have already mentioned, the input model sets should cover the whole transformation. Therefore, the main goal of the approach is to minimize the possibility that the transformation works perfectly for N input models, but fails for the (N+1)th input model. Or what is even worse: the output is generated for the (N+1)th input model, but the output is not the expected one; i.e., there is a conceptual error within the transformation definition.

Fig. 5 introduces the architecture of the approach. Input model sets are automatically generated based on the input metamodel (metamodel A), and the transformation processes transform the input models. The output models should instantiate the output metamodel (metamodel B). Finally, domain specialists verify the correspondence between the input and output models (corresponds).

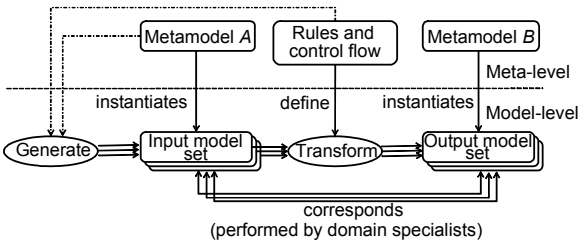

Fig. 5 Architecture of the test-driven validation approach

Scenarios that are targeted to be supported by the test-driven validation approach are as follows:

1. Automatic generation of valid input models that support the testing of the model transformation. The generation is based on the metamodel of the input domain and the transformation definition (control flow model and the transformation rules).

2. Automatic generation of valid input model sets that cover the whole model transformation, i.e., executing the transformation with an input model set means that all of the transformation rules will be executed, and all of the paths in the control flow model are traversed.

3. Automatic generation of a valid and minimal input model set that covers the whole model transformation.

4. Automatic generation of valid input models that support the testing of one or more selected transformation rules, i.e., executing the transformation with these input models means that the selected transformation rules will be executed. Other transformation rules of the control flow model can be skipped in this scenario; e.g., certain branches or loops of the whole transformation can be omitted. The main goal of this scenario is the debugging of the selected transformation rules.

5. Automatic generation of valid and minimal input models that support the testing of one or more selected transformation rules, e.g., a selected sequence of transformation rules within the whole transformation definition.

Addressing the above scenarios, the test-driven validation approach can effectively support the verification/validation of graph rewriting based model transformations. As highlighted in Section 2, there is no question that testing software artifacts, including model transformations, is costly, but not testing model transformations is even more expensive.

During the analysis and implementation of the above scenarios, we have to consider the following elements and aspects of model transformations and transformation rules:

1. To cover all of the transformation rules, all LHS patterns should either be present in the generated input model or be established during the transformation execution before reaching the rule that requires the pattern.

2. In the method the modifications performed by the rules should be considered. Rules can also delete or break LHS patterns prepared for other rules. Also,

rules can prepare LHS patterns for other rules executed later. Therefore, deletion and creation of nodes and edges, furthermore the attribute value modification, should also be considered in at least the advanced version of the solution.

3. The control flow model of the model transformation has an effect on the processing. Not only rule sequences but the effects of the conditional branches and the loops should be considered.

4. Different handling is required by the in-place transformations and the transformations generating a separate output model. We should know whether the transformation modifies the input model. Furthermore, we should consider that if the transformation generates a separate output model, does it modify the input model as well?

5. The generated input model is ideally connected, but it is not a strict requirement. This depends on the actual domain and the metamodel of the domain.

We have worked out the concept of two versions of the test-driven validation approach, basic and advanced.

The basic algorithm considers the following aspects of model transformation definitions: (1) transformation rules that should be covered by the generated input model; (2) LHS structure of the concerned rules.

The advanced solution extends it with the following considerations:

1. Collects the RHS patterns of the processed rules in a global store, and considers both the actually generated input model and the RHS patterns of the already processed rules when it decides whether the LHS pattern of the next rule can be present in the model at a certain point of model processing.

2. Takes into account rule sequences and their operations (node and edge deletion, creation and attribute modifications).

(i) The solution applies rule concatenations to calculate the resulting RHS patterns at a certain point of the transformation. Rule concatenation means contracting two rules to derive one transformation rule whose behavior functionally replaces the application of the two original rules. The concatenation results in a new rule with a new LHS and RHS pattern. The calculated RHS pattern is also considered when the method searches the LHS of the next patterns.

(ii) The solution includes the conditional branches, and therefore considers the possible execution paths of the transformation. This is also supported by rule concatenation, and can result in different rule execution sequences.

(iii) The solution takes into account the loops of the control flow definition. Loops can also result in different execution paths and thus have an effect on the result of the rule execution sequences; therefore, they can also result in different patterns in the processed model.

The GENERATESINPUT-BASIC algorithm provides the transformation definition, the collection of the concerned rules, and the input metamodel as parameters. It initializes a model based on the input metamodel. This model is built by the next part of the algorithm. The core of the algorithm is a loop that takes the next transformation rule based on the control flow model of the transformation and the collection of the rules that should be covered by the generated input model. Next, the algorithm checks if the LHS of the actual rule is already present in the generated model. If not, then it clones the LHS and attaches the copy of the LHS to the input model under generation. This method, attaching the LHS of the actual rule, can occur in different ways. In the case of the basic algorithm, we search for a common node based on the meta type of the node, and attach the new pattern using this common point. Necessarily, this step considers the rules of the input metamodel in order for the generated model to be a valid instance of the metamodel.

Algorithm 1 GENERATESINPUT-BASIC

```
GENERATESINPUT-BASIC(Transformation T, Collection
    RuleCollection, Model InputMetamodel): Model
1 Model InputModel=INITIALIZEMODEL(InputMetamodel)
2 while (Rule rule=T.GetNextRule(RuleCollection)) do
3   if not InputModel.ContainsPattern(rule.LHS) then
4     Model temporaryPattern=CLONEMODEL(rule.LHS)
5     InputModel.AddStructure(temporaryPattern)
6   end if
7 end while
8 return InputModel
```

The computational complexity of the GENERATESINPUT-BASIC algorithm for generating valid input models that support the testing of the model transformation is $O(\sum_{1...k}(v_k^2+v_k*v_m))$, where k is the number of rules in RuleCollection, v_k is the number of

vertices in the kth transformation rule of RuleCollection, and v_m is the number of vertices in the metamodel of the generated model, i.e., the metamodel of the input model for transformation T. The first part of the sum stands for pattern matching (ContainsPattern) and the second part for the appropriate model concatenation (AddStructure).

The GENERATESINPUT-ADVANCED algorithm extends the basic algorithm with the following steps:

1. It stores the RHS patterns of the processed transformation rules in RHS-Store. Furthermore, the LHS of the actual rule is searched not only in the actual version of the generated model, but also in RHS-Store.

The CALCULATERHSPATTERNVARIATIONS method applies the rule concatenation technique and calculates the different RHS pattern variations. The method gets the transformation, the actual rule, and the RHS patterns from RHS-Store to use them during the calculation of the pattern variations.

The CALCULATERHSPATTERNVARIATIONS method also considers both the conditional branches and the loops of the transformation definition.

These techniques of the GENERATESINPUT-ADVANCED algorithm make it possible to generate minimal model sets that support the testing of the whole transformation. This means that the techniques help to minimize the number and the size of the generated models.

Algorithm 2 GENERATESINPUT-ADVANCED

```
GENERATESINPUT-ADVANCED(Transformation T, Collection
RuleCollection, Model InputMetamodel): Model
1  Model InputModel=INITIALIZEMODEL(InputMetamodel)
2  PatternStore RHS-Store=INITIALIZEPATTERNSTORE()
3  while (Rule rule=T.GetNextRule(RuleCollection)) do
4    if not InputModel.ContainsPattern(rule.LHS) && not
       RHS-Store.ContainsPattern(rule.LHS) then
5      Model temporaryPattern=CLONEMODEL(rule.LHS)
6      InputModel.AddStructure(temporaryPattern)
7      RHS-Store.AddPattern(rule.RHS)
8      Pattern[] RHS-PatternVariations=CALCULATERHSPAT-
       TERNVARIATIONS(T, rule, RHS-Store)
9      RHS-Store.AddPatterns(RHS-PatternVariations)
10   end if
11 end while
12 return InputModel
```

The complexity of the GENERATESINPUT-ADVANCED algorithm for generating valid input models that support the testing of the model transformation is $O(\sum_{1...k}(2*v_k^2+v_k*v_m+k*v_k^2))=O(\Sigma_{1...k}((2+k)*v_k^2+v_k*v_m))$. The first part of the sum stands for pattern matching (ContainsPattern), the second part for the appropriate model concatenation (AddStructure), and the third part for RHS pattern variation calculation (CALCULATERHSPATTERNVARIATIONS). Finally, the sum is consolidated.

The presented algorithms address the above requirements; i.e., applying these algorithms we can automatically generate valid input models that support the testing of one or more selected transformation rules. Using these algorithms with different input parameters, we can also generate valid and minimal input model sets that cover whole model transformations. The details of certain parts of the algorithms, e.g., the 'get next' rule of the transformation (taking into account the branches and the loops), the pattern search in the generated model and in RHS-Store, and the CALCULATERHSPATTERNVARIATIONS method, can be implemented in different ways. This also means that further optimization can be introduced, e.g., with the application of different heuristics.

6 Open issues in the field of verification/validation of model transformations

In our terminology, a model transformation is a program that processes graph-based models. The operation of such transformations is based on the theory of graph rewriting. Based on the current capabilities of the available approaches and tools, and also considering the results available in the field, we identified the following open issues as challenges related with the verification/validation of model transformations. We believe that solving these issues will significantly improve the usability and availability of model transformation based approaches.

1. Verification and validation of global properties. The scope of a property can be either local or global. With the exception of the model checking approach, one of the main limitations of the current approaches and tools is the local nature of their transformation rules. Local nature of a property means that if we aim to specify a constraint for an element, it must be included in the context of a transformation rule, or must be referenced by a traversal expression assigned to a rule element. Elements not appearing in a rule cannot be included in

the verification/validation expressions. Therefore, this method does not provide an easy solution to checking constraints of a global nature (e.g., deadlock examination). Of course, there are numerous cases, for example, source code generation from a state-chart model, user interface generation from a resource model, or projecting a source model into a different domain, in which the entire right side is generated. Thus, all the output model elements are included in transformation rules.

Approaches should be developed that support the verification/validation of global properties in the processed models.

2. High-level languages/methods to define the verification/validation properties. Generating source code from software models is a widely used method to make system development more effective. While generating model artifacts we can require rather usable quality factors, but in the case of source code generation, functional and complex source code properties still cannot be defined. There are approaches (e.g., Fujaba, VIATRA2, and VMTS) that facilitate source code generation, but they are language-specific (they can process only a few types of source languages) (Fujaba), or the verification/validation opportunities of such transformations are at too low levels. Low-level verification/validation capabilities mean that even a short source code requires a relatively large model, e.g., an abstract syntax tree model; the transformation designer must be familiar with all of the details regarding the generated source code to be able to define its quality requirements. On the other hand, the goal is to involve a wider range of users group in order to provide their quality-related verification/validation properties. High-level, easy-to-use languages should be provided that facilitate defining verification/validation requirements against model transformation.

Currently, the most user-friendly languages are OCL, Pyton, Java, and similar languages that can be used to define the requirements. The research activities should identify the appropriate, first-order logic, second-order logic, or other formalism. These languages should be general-purpose languages and easy to use even for novice users.

3. Supporting the verification/validation of domain-specific properties. Transformation methods should be provided that are able to verify/validate domain-specific properties using model transformations. These are output, model-related requirements that model transformations should support. In several cases, model transformation rules do not contain certain nodes or edge types that we seek to include in our verification/validation requirements. These requirements may relate to the temporary (during model processing) or final (after model processing) state of the processed or generated models. There exist many different directions that can be taken; e.g., we can state additional requirements against the input and output models (metamodel constraints), or the model transformations can be automatically extended with appropriate testing and validating transformation rules.

4. Compositionality conditions. Model transformation related compositionality conditions should be developed: if we can prove that certain elements of a model transformation are correct, then what further conditions are required for the whole transformation to be free of conceptual errors?

5. Automatic identification of properties to be verified/validated. Algorithms should be developed that facilitate us to automatically identify (project or metamodel-specific) model properties that should be verified/validated.

6. Verification-related spatial and time complexities. The verification-related spatial and time complexities should be addressed. Most of the verification approaches require significant computational capacity and a considerable amount of time to be executed. These complexities, and thus the complexity of the verification process, should be reduced.

7. Applicability of the existing approaches. The applicability of the already existing approaches and tools should be tested within industrial environments: experiments should be performed on larger-scale (industrial size) transformations and models.

8. Industrial model checkers. Model-checker tools should be built with such properties that make it possible to be applicable to the industry. For example, model-checker tools should support the results of the performed verifications that will be automatically propagated back to the original domain.

9. Analysis patterns for static verification methods. Static verification means that only the definition of model transformation is used during the analysis and no concrete input models are taken into

account. Hence, the results are valid for all possible output models and the analysis has to be performed only once. However, the disadvantage of this method is the complexity of the analysis itself. The static verification of all attributes is not possible in general, since, e.g., the termination itself, is undecidable in general (Plump, 1998). However, it would be beneficial to collect some special cases when the verification can always provide a result, for example, if certain design rules are applied during the implementation of the transformations. A promising method that would improve the verification solutions is the use of model transformation analysis patterns. These would be design patterns that should be used during the implementation of model transformations. The use of a pattern would assure that certain properties are true for the selected part of the model transformations. These patterns should be well documented and provided in a standard catalogue form like the classical design patterns.

According to the current state of the art, these open issues can seem daunting. Each of them requires further research and development. Some model transformation approaches and/or tools partly address one or two of these open issues, but most of these tools are used only within academia and among research groups. Therefore, the most important challenge is to ensure that these verification and validation approaches become applicable within industrial environments.

7 Conclusions

Different semantic information can be lost or misinterpreted in a transformation due to errors in the definition of the transformation or in the processing method. Methods are required to verify that the semantics used during the analysis are indeed preserved across the transformation. Automatic model processing certainly increases the quality of model transformations as errors are not added by accident into transformation definitions. Verification and validation of model transformations is required, which assures that conceptual errors in model transformations do not remain hidden.

This paper has emphasized the necessity of verification/validation methods which increase the quality of model transformations and help to ensure that model transformations perform what they are intended to do. Focusing on graph rewriting based model transformations, we have discussed the different scenarios of model transformation verification and validation. Next, we have provided our dynamic validation method, and introduced the key motivation and challenging points of the test-driven validation approach. Also, we have provided both the basic and advanced versions of our solution, which make it possible to automatically generate test input models for model transformations. Finally, we have compiled the actual open issues in the field of verification/validation of model transformations. We believe that addressing these issues will significantly improve the capabilities and application of model transformation verification/validation methods and tools.

References

Akehurst, D., Kent, S., 2002. A relational approach to defining transformations in a metamodel. *LNCS*, **2460**:243-258. [doi:10.1007/3-540-45800-X_20]

Amrani, M., Dingel, J., Lambers, L., *et al.*, 2012. Towards a model transformation intent catalog. Proc. 1st Workshop on the Analysis of Model Transformations, p.3-8. [doi:10.1145/2432497.2432499]

Assmann, U., 1996. How to uniformly specify program analysis and transformation with graph rewrite systems. *LNCS*, **1060**:121-135. [doi:10.1007/3-540-61053-7_57]

Assmann, U., Ludwig, A., 2000. Aspect weaving with graph rewriting. *LNCS*, **1799**:24-36. [doi:10.1007/3-540-40048-6_3]

Asztalos, M., Lengyel, L., Levendovszky, T., 2010a. Towards automated, formal verification of model transformations. IEEE Int. Conf. on Software Testing V&V, p.15-24.

Asztalos, M., Ekler, P., Lengyel, L., *et al*., 2010b. Applying online verification of model transformations to mobile social networks. Electronic Communications of the EASST. Proc. 4th Int. Workshop on Graph-Based Tools.

Blostein, D., Fahmy, H., Grbavec, A., 1996. Issues in the practical use of graph rewriting. *LNCS*, **1073**:38-55. [doi:10.1007/3-540-61228-9_78]

Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. *IBM Syst. J.*, **45**(3):621-646. [doi:10.1147/sj.453.0621]

de Lara, J., Taentzer, G., 2004. Automated model transformation and its validation using AToM^3 and AGG. *LNCS*, **2980**:182-198. [doi:10.1007/978-3-540-25931-2_18]

de Lara, J., Vangheluwe, H., Alfonseca, M., 2004. Meta-modelling and graph grammars for multiparadigm modelling in AToM. *Softw. Syst. Model.*, **3**(3):194-209. [doi:10.1007/s10270-003-0047-5]

Ehrig, H., Habel, A., Kreowski, H.J., *et al*., 1991a. From graph grammars to high level replacement systems. *LNCS*, **532**:269-291. [doi:10.1007/BFb0017395]

Ehrig, H., Habel, A., Kreowski, H.J., *et al*., 1991b. Parallelism and concurrency in high-level replacement systems. *Math. Struct. Comput. Sci.*, **1**(3):361-404.

Ehrig, H., Engels, G., Kreowski, H.J., *et al*. (Eds.), 1999. Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools. World Scientific, Singapore.

Ehrig, H., Habel, A., Padberg, J., *et al*., 2004. Adhesive high-level replacement categories and systems. *LNCS*, **3256**:144-160. [doi:10.1007/978-3-540-30203-2_12]

Ehrig, H., Ehrig, K., Prange, U., *et al*., 2006. Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer.

Giese, H., Glesner, S., Leitner, J., *et al.*, 2006. Towards verified model transformations. ModeVVa Workshop Associated to MODELS, p.78-93.

Guerra, E., de Lara, J., 2007. Event-driven grammars: relating abstract and concrete levels of visual languages. *Softw. Syst. Model.*, **6**(3):317-347. [doi:10.1007/s10270-007-0051-2]

Habel, A., Heckel, R., Taentzer, G., 1996. Graph grammars with negative application conditions. *Fundam. Inform.*, **26**:287-313.

Hetzel, W.C., 1998. The Complete Guide to Software Testing (2nd Ed.). Wiley.

Kaner, C., 2006. Exploratory testing. Quality Assurance Institute Worldwide Annual Software Testing Conf.

Kaner, C., Falk, J., Nguyen, H.Q., 1990. Testing Computer Software (2nd Ed.). Wiley, New York.

Karsai, G., Agrawal, A., Shi, F., *et al*., 2003. On the use of graph transformation in the formal specification of model interpreters. *J. Univ. Comput. Sci.*, **9**(11):1296-1321.

Kolawa, A., Huizinga, D., 2007. Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press, p.41-43.

Lack, S., Sobocinski, P., 2004. Adhesive categories. *LNCS*, **2987**:273-288.

Leitner, A., Ciupa, I., Oriol, M., *et al.*, 2007. Contract Driven Development = Test Driven Development – Writing Test Cases. Proc. 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering, p.425-434. [doi:10.1145/1287624.1287685]

Lengyel, L., 2006. Online Validation of Visual Model Transformations. PhD Thesis, Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary.

Mens, T., van Gorp, P., 2006. A taxonomy of model transformation. Proc. Int. Workshop on Graph and Model Transformation, p.125-142.

Narayanan, A., Karsai, G., 2008. Towards verifying model transformations. *Electron. Notes Theor. Comput. Sci.*, **211**:191-200. [doi:10.1016/j.entcs.2008.04.041]

OMG, 2010. Unified Modeling Language (UML) Specification, Version 2.3, OMG document formal/2010-05-03, Available from http://www.uml.org/.

OMG, 2011. OMG Query/View/Transformation (QVT) Specification, Meta Object Facility 2.0 Query/Views/Transformation Specification. OMG doc. formal/2011.01.01. Available from http://www.omg.org/spec/QVT/.

OMG, 2012. OMG Object Constraint Language (OCL) Specification, Version 2.3.1. OMG Document Formal/2012-05-09. Available from http://www.omg.org/spec/OCL/.

OMG, 2014. OMG Model-Driven Architecture (MDA) Specification. OMG Document ormsc/14-06-01. Available from http://www.omg.org/mda/.

Pan, J., 1999. Software Testing - 18-849b Dependable Embedded Systems. Carnegie Mellon University. Available from http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/.

Plump, D., 1998. Termination of graph rewriting is undecidable. *Fundam. Inf.*, **33**(2):201-209.

Rozenberg, G. (Ed.), 1997. Handbook on Graph Grammars and Computing by Graph Transformation: Foundations. World Scientific, Singapore.

Schürr, A., 1994. Specification of graph translators with triple graph grammars. *LNCS*, **903**:151-163. [doi:10.1007/3-540-59071-4_45]

Syriani, E., 2009. Matters of Model Transformation. No. SOCS-TR-2009.2, School of Computer Science, McGill University.

Taentzer, G., Ehrig, K., Guerra, E., *et al.*, 2005. Model transformation by graph transformation: a comparative study. ACM/IEEE 8th Int. Conf. on Model Driven Engineering Languages and Systems, p.1-48.

Vajk, T., Kereskényi, R., Levendovszky, T., *et al*., 2009. Raising the abstraction of domain-specific model translator development. 16th Annual IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems, p.31-37. [doi:10.1109/ECBS.2009.30]

van Gorp, P., Stenten, H., Mens, T., *et al*., 2003. Towards automating source-consistent UML refactorings. *LNCS*, **2863**:144-158. [doi:10.1007/978-3-540-45221-8_15]

Varró, D., Pataricza, A., 2003. Automated formal verification of model transformations. Proc. UML03 Workshop, p.63-78.

Varró, D., Varró-Gyapay, S., Ehrig, H., *et al.*, 2006. Termination analysis of model transformations by Petri nets. *LNCS*, **4178**:260-274. [doi:10.1007/11841883_19]