Electronic Communications of the EASST
Volume 54 (2012)

Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)

Integration of Triple Graph Grammars and Constraints

Stephan Hildebrandt, Leen Lambers, Basil Becker, Holger Giese

12 pages

Guest Editors: Christian Krause, Bernhard Westfechtel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Integration of Triple Graph Grammars and Constraints

Stephan Hildebrandt[1], **Leen Lambers**[1], **Basil Becker**[1], **Holger Giese**[1]

[1] surname.lastname@hpi.uni-potsdam.de
Hasso Plattner Institute at the University of Potsdam*

Abstract: Metamodels are often augmented with additional constraints that must be satisfied by valid instances of these metamodels. Such constraints express complex conditions that cannot be expressed in the metamodel itself. Model transformations have to take such constraints of the source and target metamodels into account. Given a valid source model, which satisfies the source constraints, a model transformation is expected to return a valid target model (*forward validity*). However, in current model transformation definition and tool support, such an integration with source and target constraints including validation mechanisms is often ignored or not satisfactory yet.

In this paper, we describe how the integration with source and target constraints can be achieved for the special case of model transformations defined by Triple Graph Grammars (TGGs). First, we extend the relational model transformation definition for TGGs and integrate it with source and target constraints. Moreover, we describe how forward/backward validity of TGGs with constraints can be automatically checked, either by static analysis using an invariant checker, or by generating and validating metamodel instances. Finally, we describe how to integrate constraints into our TGG-based model transformation implementation and automatic conformance testing framework.

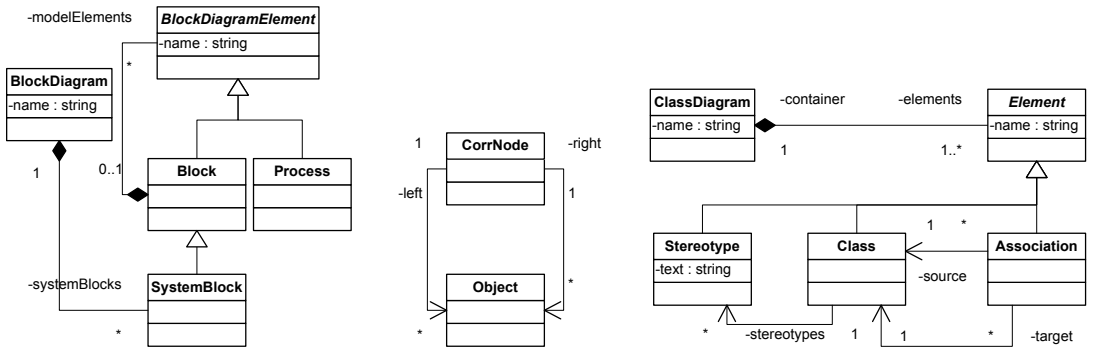
Keywords: Model Transformation, Constraints, Test Case Generation, Invariant Checking

1 Introduction

Model-Driven Engineering puts models and model transformations into the focus of the development process. In practice, models are mostly defined using metamodels, which specify the kinds of elements that can be used in a model of that type. Metamodels are often augmented with constraints, which models must satisfy to be valid instances of a metamodel. Such constraints are complex conditions that cannot be expressed in the metamodel itself. Modeling tools have to consider these constraints in order to correctly work with models. In particular, model transformation tooling, which plays a key role in Model-Driven Engineering, needs to consider constraints as well.

A model transformation is expected to always return a valid target model if it is provided with a valid source model. *Valid* means that the models not only adhere to the structure defined

* This work was developed in the course of the project - Correct Model Transformations - Hasso Plattner Institut, Universität Potsdam and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft. See http://www.hpi.uni-potsdam.de/giese/projekte/ kormoran.html?L=1.



(a) BlockDiagram metamodel (b) Correspondence metamodel (c) ClassDiagram metamodel

Figure 1: Example metamodels

by their metamodels, but also satisfy the metamodels' constraints. We call transformations that always produce valid target models for valid source models *forward valid* transformations. The definition of *backward valid* is analogous for bidirectional transformation approaches.

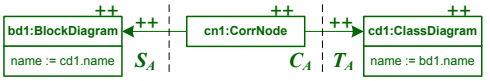
However, existing implementations mostly do not consider constraints defined on metamodels at all. Instead, the problem is shifted to the designer of the transformation specification who is expected to define appropriate pre- and postconditions filtering out invalid models, which often resemble constraints already defined in the metamodels. It remains an open question how to ensure consistency between constraints in the metamodel and pre- and postconditions of the transformation specification defined by the designer.

For these reasons, we have extended our existing tools for model transformation [GHL12] and conformance testing of model transformations [HLG+12] based on Triple Graph Grammars [Sch94] to consider metamodel constraints, focusing on design-time checks as well as runtime checks revealing problems with forward or backward validity of the transformation. We describe the integration of TGGs with constraints in Section 3 and present two approaches to check the forward/backward validity of forward/backward transformations derived from a TGG in Section 4, one based on static analysis using an invariant checker, and another one based on counter example generation. Section 5 explains how we extended our conformance testing framework and our model transformation implementation to support constraints. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

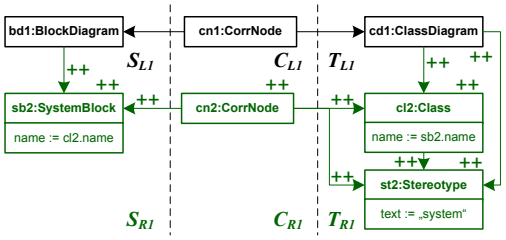
2 Our Triple Graph Grammar Framework

Triple Graph Grammars [Sch94] are an important representative of relational model transformation specifications. To illustrate the following explanations, we will use a model transformation from simple SDL block diagrams[1] to UML class diagrams. The metamodels of both modeling languages are shown in Figure 1. Block Diagrams are a hierarchical structure of *Blocks*. *Blocks* can be nested and the topmost blocks are *SystemBlocks*. In addition, *Blocks* can contain *Pro-*

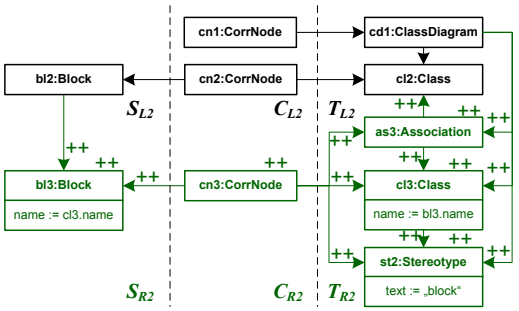
[1] A simplified version of SDL block diagrams (http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf).



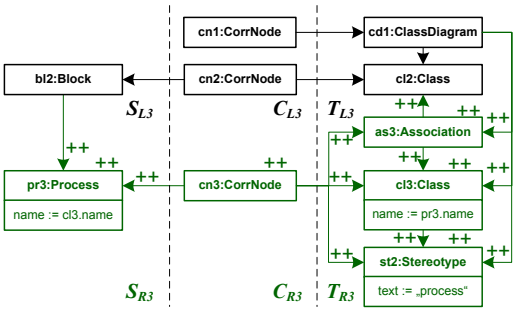
(a) Axiom Rule (*BlockDiagram* to *ClassDiagram*)



(b) Rule 1 (*SystemBlock* to *Class*)



(c) Rule 2 (*Block* to *Class*)



(d) Rule 3 (*Process* to *Class*)

Figure 2: Example triple graph grammar

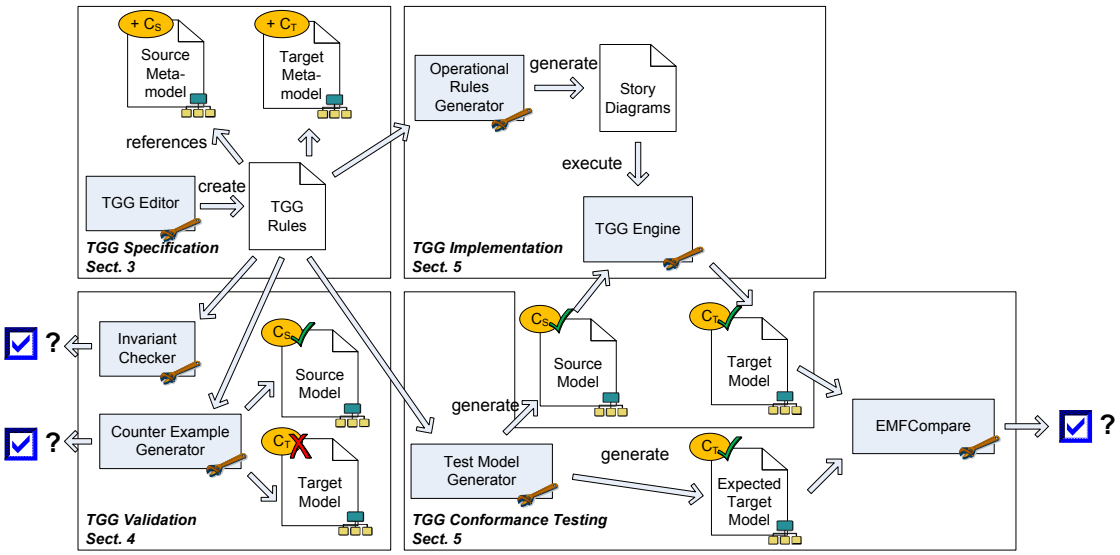


Figure 3: Triple Graph Grammar tool framework

cesses, which cannot contain any elements. Class Diagrams can contain *Classes*, *Stereotypes*, which are attached to a *Class*, and *Associations*, which connect two classes.

We have implemented a framework based on TGGs for model transformation and synchronization, as well as for conformance testing of TGG model transformations. It is based on the Eclipse Modeling Framework (EMF)[2] and can be downloaded from our Eclipse update site[3]. The framework's architecture is shown in Figure 3. We repeat the basic principles here and refer to [GHL12, GNH10, HLG+12] for more detailed information.

The *Source* and *Target Metamodels*, S_{TT} and T_{TT}, respectively, can be edited using the existing Ecore metamodel editor. *TGG Rules* can be created and edited using the *TGG Editor*. The TGG rules of the transformation between Block Diagrams and Class Diagrams are shown in Figure 2. Each rule consists of three domains: A source domain on the left, which contains elements of the Block Diagram, a target domain on the right, which contains elements of the Class Diagram, and a correspondence domain in the middle, which contains the so-called correspondence model. The correspondence model explicitly stores correspondence relationships between source and target model elements. Its metamodel C_{TT} is shown in Figure 1b. A *CorrNode* maps arbitrary objects in the source model to objects in the target model. These three models form a so-called *triple graph*, denoted by *SCT*, where *S* denotes the source, *C* the correspondence, and *T* the target component. The notation used in Figure 2 combines the left-hand side (LHS) and right-hand side (RHS) of a graph transformation rule. Elements contained in the LHS and RHS are black, elements contained in the RHS only are drawn green and marked with "++".

Like ordinary graph grammars, Triple Graph Grammars have a start graph, called axiom[4], denoted by $S_AC_AT_A$. The axiom creates the root nodes of all three models. The remaining TGG rules create the other elements of the models in certain contexts: Rule 1 links a *SystemBlock* to an existing *BlockDiagram* and a *Class* to the corresponding *ClassDiagram*. Likewise, rules 2 and 3 create *Blocks* and *Processes* in existing *Blocks* and a corresponding pattern of a *Class*, *Association*, and *Stereotype* in the *ClassDiagram*. More formally, a *triple graph grammar* (TGG) consists of a set of triple graph rules $\mathscr{R}$ typed over $S_{TT}C_{TT}T_{TT}$ (the *type graph* or metamodel, which results by connecting the source and target metamodels via their correspondences) and a triple start graph $S_AC_AT_A$, called axiom, also typed over $S_{TT}C_{TT}T_{TT}$. For more information about the formalization of TGGs, we refer to [GHL12].

As it is, the TGG can be used to create all three models in parallel. The basic principle is depicted in Figure 4, where the nodes represent triple graphs and the arrows represent TGG rule applications. First, the axiom rule is applied once and creates $S_AC_AT_A$. A randomly selected TGG rule is then applied to create, e.g., $S_{13}C_{13}T_{13}$. Then, another rule is selected and applied to create, e.g., $S_{26}C_{26}T_{26}$. With $S_iC_iT_i \Rightarrow S_{i+1}C_{i+1}T_{i+1}$ we denote a rule application from $S_iC_iT_i$ to $S_{i+1}C_{i+1}T_{i+1}$ and we write $S_GC_GT_G \overset{*}{\Rightarrow} S_{G'}C_{G'}T_{G'}$ to denote 0 to a random number of rule applications from $S_GC_GT_G$ to $S_{G'}C_{G'}T_{G'}$. All triple graphs *SCT* that can be created accordingly from a TGG belonging to $\mathscr{L}(tgg)$, the TGG language. More formally, given a particular $tgg = (S_AC_AT_A, \mathscr{R})$, typed over $S_{TT}C_{TT}T_{TT}$, then the *triple graph language* $\mathscr{L}(tgg)$ consists of all triple graphs $S_GC_GT_G$ such that $S_AC_AT_A \overset{*}{\Rightarrow} S_GC_GT_G$ via rules in $\mathscr{R}$. The *Test Model Generator*

[2] http://www.eclipse.org/modeling/emf

[3] http://www.mdelab.de/update-site

[4] In particular, we use a so-called axiom rule, which is applied once to the empty graph and thereby sets correct attribute values creating the concrete axiom.

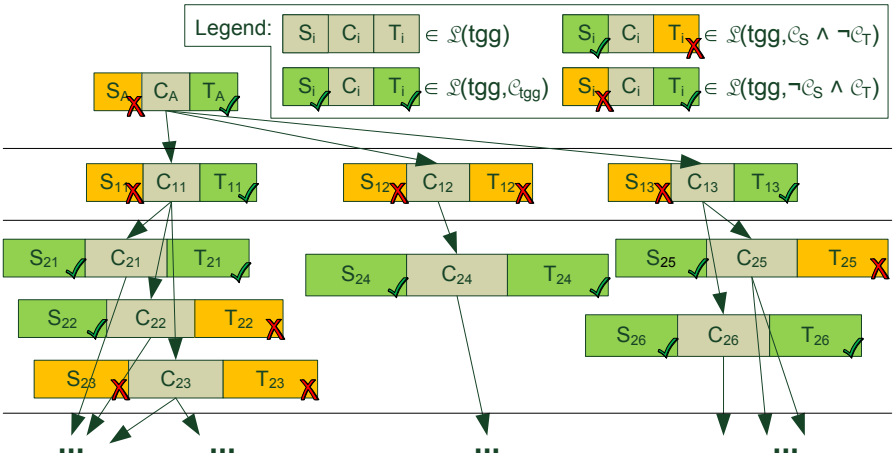


Figure 4: Step-wise derivation of triple graphs using a TGG with constraints

(cf. Figure 3) uses this principle to generate a *Source Model* and an *Expected Target Model* that can be used as test input and test oracle to automatically verify conformance of a TGG with the corresponding TGG implementation. To test the forward transformation, the *Source Model* (S_i of $S_iC_iT_i$) is input to the *TGG Implementation* under test. The *TGG Implementation* outputs a *Target Model*, which is compared to the *Expected Target Model* (T_i of $S_iC_iT_i$) using *EMFCompare*. If both models are equal, the *TGG Implementation* has passed the conformance test.

To actually perform a model transformation using TGGs, operational rules have to be derived for the three different transformation directions supported by TGGs: A *forward transformation* takes a model of the left domain, a Block Diagram in the example, and produces the other two models. A *backward transformation* is the opposite, a model of the right domain is transformed to a model of the left domain and a correspondence model. A *mapping transformation* takes two existing models of both domains and produces only the correspondence model. For each of these directions, separate operational rules have to be derived. In our model transformation tool, *Story Diagrams*[5] describe the operationalized form of the *TGG Rules*. Operational rules add the elements of the source model domain of the respective direction to their LHS, e.g., forward rule 1 (cf. Figure 2b) includes *sb2* in its LHS. It does not create a new *SystemBlock*. In this paper, we focus on the forward direction. The backward direction is always analogous.

In practice, the *metamodels* of source and target language are often augmented with a set of constraints $\mathcal{C}_S$ and $\mathcal{C}_T$, respectively. Our existing framework did not support such constraints. Therefore, we extended the formal definition of TGGs with constraints, added an automatic checking mechanism for forward/backward validity TGGs with constraints based on two tool components (*Invariant Checker* and *Counter Example Generator* in Figure 3), and extended our TGG model transformation implementation as well as automatic conformance testing framework with support for constraints. These conceptual extensions and the corresponding new tool components are presented in the next sections.

[5] Story Diagrams are executable models consisting of a combination of UML Activity Diagrams with graph transformation.

3 Integration of TGG Specifications and Constraints

As mentioned before, metamodels are often augmented with constraints. Such constraints can be as simple as multiplicities of references but also complex conditions spanning multiple elements of the metamodel are possible. The *Object Constraint Language* (OCL) is the commonly used language to define metamodel constraints.

The example metamodels in Figure 1 already contain multiplicities: Trivial examples are the container ends of containment references, e.g., *BlockDiagram* and *ClassDiagram*, which always have a lower and upper bound of 1. But also several more complex constraints can be defined. In Class Diagrams, all *Stereotype*s of a *Class* must have different values in their *text* attributes. This is expressed in OCL as follows:

```
(C1) context Class inv UniqueStereotypes:
self.stereotypes->forAll(e1|self.stereotypes->forAll(e2|e1 = e2 or e1.text <> e2.text))
```

For Block Diagrams, we have the following constraints: First, a *SystemBlock* may not contain other *SystemBlock*s or *Process*es. In OCL, this is expressed like this:

```
(C2) context SystemBlock inv NoSystemBlocksOrProcesses:
not self.modelElements->exists(e|e.oclIsKindOf(SystemBlock) or e.oclIsKindOf(Process))
```

Second, each *Block* in a Block Diagram may either contain other *Block*s or *Process*es, but not both. If a *Block* contains other *Blocks*, then there must be at least two contained *Block*s. This constraint is expressed in OCL as follows:

```
(C3) context Block inv BlockHierarchyConstraint:
self.modelElements->isEmpty() or
self.modelElements->forAll(e|e.oclIsKindOf(Process)) or
(self.modelElements->forAll(e|e.oclIsKindOf(Block)) and self.modelElements->size() >= 2)
```

Triple Graph Grammars as presented in Section 2 do not consider such metamodel constraints yet. Considering metamodel constraints in a TGG essentially means restricting the set of triple graphs SCT that can be created with the TGG to those that also satisfy the constraints $\mathscr{C}_S$ and $\mathscr{C}_T$ typed over the source and target metamodels S_{TT} and T_{TT}, respectively.[6] Given a so-called TGG constraint $\mathscr{C}_{tgg} = \mathscr{C}_S \wedge \mathscr{C}_T$ for a $tgg = (S_A C_A T_A, \mathscr{R})$, being typed over $S_{TT} C_{TT} T_{TT}$, then the *triple graph language with constraints* $\mathscr{L}(tgg, \mathscr{C}_{tgg})$ consists of all triple graphs $S_G C_G T_G \models \mathscr{C}_{tgg}$ such that $S_A C_A T_A \overset{*}{\Rightarrow} S_G C_G T_G$ via rules in $\mathscr{R}$. We say that a model is *valid* w.r.t. a metamodel with constraints, if it is typed over the metamodel and, additionally, satisfies all its constraints.

Figure 4 describes a TGG language with constraints. Every node of the tree is a triple graph $S_i C_i T_i$ that can be generated by tgg, i.e., $S_i C_i T_i \in \mathscr{L}(tgg)$. However, not all of them also satisfy the constraints of the source and target metamodels. In Figure 4, the source and target components S_i and T_i that are valid w.r.t. source and target constraints are marked with check marks to symbolize satisfied constraints. In contrast, invalid components are marked with crosses to symbolize violated constraints. Triple graphs, where the source or target component is valid and marked with a check mark, and the target or source component is not valid and marked with a cross, belong to $\mathscr{L}(tgg, \mathscr{C}_S \wedge \neg\mathscr{C}_T)$ or $\mathscr{L}(tgg, \mathscr{C}_T \wedge \neg\mathscr{C}_S))$, respectively. But only those triple graphs, where both components are valid w.r.t source and target constraints, belong to $\mathscr{L}(tgg, \mathscr{C}_{tgg})$.

[6] For space reasons, we do not consider correspondence metamodel constraints here, but these can be handled analogously. Moreover, models and constraints typed over S_{TT} and T_{TT} are automatically also typed over $S_{TT} C_{TT} T_{TT}$.

4 Automatic Checking of TGGs with Constraints

Recall that we say that model transformations are *forward valid* if they produce valid target models from valid source models. We want to automatically check for a tgg with constraints, whether it is *forward valid* (or *backward valid*, which is symmetric): Is there any $SCT \in \mathscr{L}(tgg)$, where $S \models \mathscr{C}_S$ and $T \not\models \mathscr{C}_T$? Or, equivalently, is there any $SCT \in \mathscr{L}(tgg, \mathscr{C}_S \wedge \neg\mathscr{C}_T)$? If not, then we have forward validity. If so, then there exists a counterexample.

We can group target constraints in $\mathscr{C}_T$ w.r.t. forward validation into the following categories: (1) Target constraints satisfied for each T in $SCT \in \mathscr{L}(tgg)$. We say that these constraints hold *by TGG construction*. (2) Target constraints satisfied for each T in $SCT \in \mathscr{L}(tgg, \mathscr{C}_S)$. (3) Target constraints not satisfied for each T in $SCT \in \mathscr{L}(tgg, \mathscr{C}_S)$. If every target constraint in $\mathscr{C}_T$ belongs to category (1) or (2), then each forward transformation derived from the tgg is forward valid.

In this section, we describe how the above question can be answered via static analysis (similar to [BLD+11], where we did this for refactorings) using our invariant checker or, dynamically, by generating explicit counterexamples.

Static Analysis The satisfaction of constraints can be invariant with respect to a set of graph transformation rules R. A constraint $\mathscr{C}$ is an *inductive invariant* of R if for all graphs G and for all rules $r \in R$, it holds that $G \models C \wedge\ G \Rightarrow_r G'$ implies $G' \models C$. We developed a static analysis technique being able to perform invariant checking [BBG+06] for constraints of the following kinds: "A specific pattern (*forbidden pattern*) should not occur in the model." or "A specific pattern (*conditional forbidden pattern*) should not occur without some other specific pattern.". Our invariant checker either reports that a constraint is indeed an invariant for a given set of rules, or it automatically computes as symbolic counterexamples all minimal situations indicating why rules might be applied to a constraint-satisfying graph leading to a violating one. Note that some of the counterexamples may represent false negatives because temporarily invalid constraints during the model transformation might be valid in the model transformation result (in case of conditional forbidden patterns) or because the invariant checker is lacking knowledge to reject counterexamples. Our invariant checker currently works with the following restrictions: it needs a flattened type graph as well as flattened rules and forbidden patterns as input [GLB+12]. Stereotypes need to be encoded by specific types. Moreover, since string attributes are not supported, we encode them by edges to data nodes as we also did in [BLD+11]. For more details to the internals of the invariant checker we refer to [BBG+06, BLD+11].

To classify target constraints into category (2) we proceed as follows: Assuming that the source and target constraints $\mathscr{C}_S$ and $\mathscr{C}_T$ have the above-described pattern form, we can apply invariant checking to verify if the target constraints $\mathscr{C}_T$ are invariant w.r.t. forward operational rules derived from a TGG, given that the source constraints $\mathscr{C}_S$ hold. Intuitively, this means that forward rules do not add any translated target structure which would violate the target constraints assuming that the source constraints hold on the source model. If the invariant checker produces at least one symbolic counterexample, which does not represent a false negative, then the constraint belongs to category (3). Constraints belong to category (1) if the invariant checker does not need the source constraints as assumed constraints to verify that the target constraints are invariants.

Consider the *UniqueStereotypes* constraint (C1), which can be formulated as a forbidden pat-

tern consisting of a Class holding two identical stereotypes. It holds by TGG construction w.r.t. forward validation, since it is satisfied in all Class Diagram models created by the example TGG (cf. Figure 2) independently of the constraints on the Block Diagram meta-model. The example TGG creates classes only together with a single stereotype. Indeed our invariant checker is able to identify via static analysis that this constraint belongs to category (1).

Constraint (C2) *NoSystemBlocksOrProcesses* is an example for category (3) w.r.t. backward validation. It can be described by two forbidden patterns, one that consists of a *SystemBlock* containing a *SystemBlock* and the other one of a *SystemBlock* containing a *Process*. The latter forbidden pattern is violated by rule 3 in the TGG because it allows to add a *Process* to a *Block*, which also includes *SystemBlocks* due to the generalization hierarchy defined in the metamodel (cf. Figure 1a). Indeed, our invariant checker will output a symbolic counterexample, representing a violation of the constraint when applying rule 3 to a *ClassDiagram*, holding one *Class* with *Stereotype* system and another associated *Class* with *Stereotype* process. Rule 2 violates the first forbidden pattern analogously. The counterexamples reported by the invariant checker may represent useful hints to the transformation developer w.r.t. repairing the transformation specification or the different levels of expressiveness of the source and target language under consideration. Defining a specific repair mechanism accordingly is part of future work.

The static analysis presented here has its limitations (e.g. constraint (C3) can not be analyzed), since only (conditional) forbidden patterns can be currently analyzed successfully by our invariant checker. Enhancing the expressiveness of our invariant checker is ongoing work. However, in [Lam10], it is described more generally how the invariant checking (or constraint preservation) problem can be reduced to the implication problem for conditions. As proven in [HP09], in the case of graphs, nested conditions are equivalently expressive to first order graph formulas. This means that the implication problem for application conditions is undecidable, in general. In practice, there is also the problem that OCL constraints have to be translated to graph constraints, which is a non-trivial task [WTEK08]. Consequently, if for these reasons static analysis is not available, we apply a dynamic analysis technique, which we present in the next section.

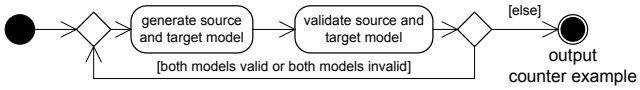


Figure 5: *Counter Example Generator* using the *Test Model Generator* (cf. Figure 3)

Generation of Counter Examples Our existing conformance testing framework can be reused to automatically check the forward/backward validity of a TGG specification with constraints. In particular, the *Test Model Generator* can be altered to search for triples *SCT*, where either the source or target model S or T is valid, but the other one is not. The behavior of this *Counter Example Generator* is depicted in Figure 5. First, a triple $SCT \in \mathscr{L}(tgg)$ is generated as before. After that, both models S and T are validated. If one model is valid and the other is invalid, a counter example was found. Otherwise, a new pair is generated.

The generator can be configured with a maximum number of iterations to avoid infinite loops, occurring because a TGG generates an infinite number of language instances in general. How-

ever, this also means that it is not guaranteed that there are no counter examples if none can be found. As part of future work, we want to use specific heuristics for a systematic model generation. This should increase the success of finding counter examples.

To actually validate the models, the EMF Validation Framework[7] is used, an extensible framework, which allows to integrate constraints specified in all kinds of languages, commonly OCL, but also Check[8] or plain Java. The actual nature of the constraints is completely transparent to clients of the validation framework. The framework lets the appropriate interpreters evaluate the constraints, e.g., the OCL interpreter for OCL constraints. In addition to constraints, multiplicities of references are also checked.

The *BlockHierarchyConstraint* belongs to category (3) as introduced in the beginning of Section 4. Consider for example rule 3 in Figure 2d), which does not check whether *Block bl2* already contains another *Block* before adding a new *Process*. Moreover, consider rule 2 (cf. Figure 2c), which is not guaranteed to be applied at least two times to make sure that a *Block* contains at least two subblocks. Our current invariant checker is not capable of checking such a complex constraint and, therefore, we applied our dynamic check directly here. The second randomly generated pair of models indeed already represented a counterexample.

5 Integration of TGG Implementation and Testing with Constraints

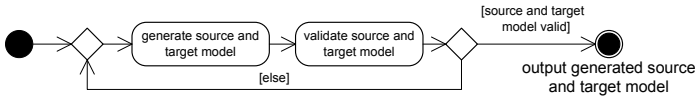


Figure 6: Extended *Test Model Generator* of the conformance testing framework (cf. Figure 3)

We also extended our conformance testing framework (cf. Figure 3), in particular the *Test Model Generator*, and our *TGG Implementation* to support constraints.

Conformance Testing: The goal of the testing framework is to check if each *SCT* obtained by a forward (backward) transformation of S (T) in the TGG implementation indeed belongs to the TGG language $\mathscr{L}(tgg)$ and the other way round. Having integrated the TGG with constraints, our new testing goal is to check if each *SCT* with valid S and T obtained by a forward (backward) transformation of S (T) indeed belongs to the constrained TGG language $\mathscr{L}(tgg, \mathscr{C}_{tgg})$. As a consequence, we focus now on generating test models satisfying the constraints $\mathscr{C}_{tgg} = \mathscr{C}_S \wedge \mathscr{C}_T$ to check conformance.[9] Consequently, in addition to our former framework, the *Test Model Generator* performs a validation on the generated source and target models (shown in Figure 6). If one of the models is invalid, a new random triple *SCT* is generated until a valid pair of source and target models S and T was found or a predefined number of misses has been reached to avoid infinite loops. The *Test Model Generator* and the *Counter Example Generator* share the same algorithm to generate triple graphs. Therefore, it is also possible to combine both components.

[7] http://www.eclipse.org/modeling/emf/?project=validation

[8] Check is provided as part of Xpand (http://www.eclipse.org/modeling/m2t/?project=xpand).

[9] We concentrate on positive testing in our testing framework, meaning that we focus on generating valid input models.

If a pair of valid models has been generated, these can be used as test input and oracle, if one of the models is invalid, these can be used as counterexamples.

TGG Implementation: The *TGG Implementation* with support for constraints is supposed to deliver a valid target model from a valid source model. It is unnecessary to execute the model transformation if the source model is already invalid. However, because of the limitations of automatically checking validity of TGGs with constraints (cf. Section 4), the TGG implementation still needs to check at runtime if a target model is valid. For these reasons, the TGG implementation has been extended with two validation steps. Before executing the transformation, the source model is validated. During the transformation, it is possible that the target model temporarily violates target constraints. Therefore, checking these constraints while the transformation is still running is not necessary. Instead, the created target model is validated after the transformation. If it is now violating target constraints, a warning is issued along with the invalid model.

The *BlockHierarchyConstraint* (C3) is an example of a constraint that may be temporarily violated during a backward transformation from a Class Diagram containing several *Class*es, which are transformed to *Block*s. After transforming the first *Class*, the constraint is violated. This constraint is also an example, where a valid source model may be transformed to an invalid target model. Considering the backward transformation (cf. Figure 2) of our running example, the following situation could occur: A Block Diagram with one *SystemBlock* containing a *Block* and a *Process* violates the *BlockHierarchyConstraint*. Its corresponding Class Diagram, however, would be valid. So, if we perform a backward transformation from that valid Class Diagram, the TGG implementation returns an invalid Block Diagram along with a warning.

6 Related Work

There is some related work concerning validation of transformation rules with consideration of constraints. A tool for the static validation of ATL transformation rules is presented in [BCG11], which considers constraints of the source and target metamodel. A transformation (meta-)model integrates the source and target (meta-)models with the execution semantics of ATL. It is possible to search for instances of the transformation metamodel that violate constraints using bounded verifiers. In contrast, we rely on a static and symbolic checking method to verify invariants given as graph constraints for a graph transformation system, which is a complete static analysis. However, constraints, usually specified in OCL, have to be translated to graph patterns first. This drawback of our approach is mitigated by the counter example generator, which generates example models and evaluates constraints directly.

To the best of our knowledge, there exist no other model transformation approaches considering the integration of metamodel constraints and allowing for automatic checking of forward/backward validity. However, most tools allow to specify additional preconditions in TGG rules using OCL, e.g., [DG09, GR12] as well as our own model transformation tool [GHL12]. In contrast, the TGG formalisms and tools presented in [GEH11, KLKS10] support TGG rule application conditions expressed as graph patterns. [DG09] and [GR12] also allow to specify postconditions in transformation rules, which are checked after the execution of a transformation rule or the complete transformation. Another approach for the verification of arbitrary model transformations based on visual transformation contracts is presented in [GLW+12]. These ap-

proaches do not investigate the interplay of additional application conditions in TGG rules with metamodel constraints, except for [KLKS10], which, however, does not focus on automatic checking mechanisms w.r.t. forward/backward validity as we do in this paper.

7 Conclusion

We have shown the importance of considering constraints defined on metamodels in TGG model transformation definition, conformance testing and implementation. Automatic checking of forward/backward validity using static analysis is merely possible for a restricted kind of constraints. Therefore, we also showed how to perform this check by generating meaningful counterexamples.

The graph patterns required as input for the invariant checker have to be derived from constraints, often expressed in OCL. This complex task [WTEK08] is currently not automated, but for a restricted part of OCL, especially simple navigation expressions, we plan to implement such a translation. Moreover, the flattening of metamodels with inheritance, rules and constraints in order to serve as input for the invariant checker still needs to be automated. It is ongoing work to improve the expressiveness of our invariant checker and to investigate also weakest pre- and post-conditions [HP09] techniques. We also want to examine how the source and target model generators can be extended with more systematic generation mechanisms. This would increase the confidence in the results of the *Counter Example Generator* as well as our conformance testing framework. The generator also needs to be integrated with the static analysis to, e.g., check if counter examples found by the static analysis are no false negatives.

Finally, after the user has found out that his transformation rules are not forward/backward valid, he needs tool support to change his transformation accordingly. However, the causes for violation of metamodel constraints may be manifold. It may be hard for the user to deduce the cause from a counter example and a violated constraint. This remains a complex open issue for future work.

Bibliography

[BBG+06] B. Becker, D. Beyer, H. Giese, F. Klein, D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of ICSE 2006*. ACM, 2006.

[BCG11] F. Büttner, J. Cabot, M. Gogolla. On validation of ATL transformation rules by transformation models. In *Proc. of MoDeVVa 2011*. ACM, 2011.

[BLD+11] B. Becker, L. Lambers, J. Dyck, S. Birth, H. Giese. Iterative Development of Consistency-Preserving Rule-Based Refactorings. In *Proc. of ICMT 2011*. LNCS 6707, pp. 123–137. Springer, 2011.

[DG09] D.-H. Dang, M. Gogolla. On Integrating OCL and Triple Graph Grammars. In Chaudron (ed.), *Models in Software Engineering*. Lecture Notes in Computer Science 5421, pp. 124–137. Springer Berlin / Heidelberg, 2009.

[GEH11] U. Golas, H. Ehrig, F. Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. *ECEASST* 39:1 – 26, 2011.

[GHL12] H. Giese, S. Hildebrandt, L. Lambers. Bridging the gap between formal semantics and implementation of triple graph grammars. *Software and Systems Modeling, Springer Berlin / Heidelberg*, pp. 1–27, 2012.

[GLB+12] H. Giese, L. Lambers, B. Becker, S. Hildebrandt, S. Neumann, T. Vogel, S. Wätzoldt. *Graph Transformations for MDE, Adaptation, and Models at Runtime*. LNCS 7320. Springer, 2012.

[GLW+12] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger. Automated Verification of Model Transformations Based on Visual Contracts. *Automated Software Engineering*, pp. 1–42, 2012.

[GNH10] H. Giese, S. Neumann, S. Hildebrandt. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In Lewerentz et al. (eds.), *Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*. LNCS 5765, pp. 555–579. Springer, 2010.

[GR12] J. Greenyer, J. Rieke. Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In Schürr et al. (eds.), *4th International Symposiom, AGTIVE 2011, LNCS 7233*. 2012.

[HLG+12] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick, I. Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In Schürr et al. (eds.), *4th International Symposiom, AGTIVE 2011, LNCS 7233*. Springer, 2012.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19, 2009.

[KLKS10] F. Klar, M. Lauder, A. Königs, A. Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Lewerentz et al. (eds.), *Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*. LNCS 5765, pp. 141–174. Springer, 2010.

[Lam10] L. Lambers. *Certifying Rule-Based Models using Graph Transformation*. PhD thesis, Technische Universität Berlin, 2010.

[Sch94] A. Schürr. Specification of graph translators with triple graph grammars. In Mayr et al. (eds.), *Proc. of the* 20^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science*. LNCS 903, pp. 151–163. Springer, June 1994.

[WTEK08] J. Winkelmann, G. Taentzer, K. Ehrig, J. M. Küster. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. *Electron. Notes Theor. Comput. Sci.* 211:159–170, 2008.