Electronic Communications of the EASST
Volume 49 (2012)

Proceedings of the
First International Workshop on
Bidirectional Transformations
(BX 2012)

Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case

Leen Lambers, Stephan Hildebrandt, Holger Giese, Fernando Orejas

16 pages

Guest Editors: Frank Hermann, Janis Voigtländer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case

Leen Lambers[1], Stephan Hildebrandt[1], Holger Giese[1], Fernando Orejas[2]

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

[2] Dpt. de Llenguatges i Sistemes Informatics Universitat Politecnica de Catalunya, Barcelona, Spain
orejas@lsi.upc.edu

Abstract: When describing bidirectional model transformations in a declarative (relational) way, the relation between structures in source and target models is specified. But not only structural correspondences between source and target models need to be described. Another important aspect is the specification of the relation between attribute values of elements in source and target models. However, most existing approaches either do not allow such a relational kind of specification for attributes or allow it only in a restricted way.

We consider the challenge of bridging the gap between a flexible declarative attribute specification and its operationalization for the triple graph grammar (TGG) specification technique as an important representative for describing bidirectional model transformations in a relational way. First, we present a formal way to specify attributes in TGG rules in a purely declarative (relational) way. Then, we give an overview of characteristic barriers that bidirectional model transformation tool developers are confronted with when operationalizing relational attribute constraints for different TGG application scenarios. Moreover, we present pragmatic solutions to overcome the operationalization barriers for different TGG application scenarios in our own TGG implementation.

Keywords: attributes, model transformation, triple graph grammars

1 Introduction

Model transformations are an important element of Model-Driven Engineering (MDE) [SK03] and allow one to automate several aspects of software development. Therefore, it is crucial that model transformations are correct and repeatable to support incremental development and maintenance of high quality software. Like programming languages, model transformation languages require an unambiguous semantics as a reference to enable the verification of the outcome, considering the model transformation specification (cf. [GGL+06]), and to ensure that different implementations result in the same outcome. In addition, an unambiguous formal semantics and

[*] This work was partially 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`.

clear understanding *how the relational (declarative) specification is operationalized* can help to identify which optimizations are really the most appropriate ones.

We consider this challenge for the specific case of triple graph grammars (TGG) [Sch94, SK08], which have a well understood formal semantics and are quite similar to other declarative and relational approaches (c.f. [GK10]) for defining bidirectional model transformations such as declarative QVT [Obj11, Ste10, Ste11]. Closing the gap between the formal semantics of TGGs and its operationalization has been subject of several work already [SK08, KLKS10, GHL12, HEGO10]. Different variants of conform and efficient operationalizations and implementations for TGGs are presented. Conformance with the TGG semantics is shown by demonstrating consistency – each transformation result of the operationalization must fit to the TGG semantics – and completeness – all possible transformations for the TGG semantics are also covered by the operationalization. However, this related work usually concentrates on how to derive such operationalizations *without taking care in detail of attribute handling*. Since attributes play an important role in today's modeling languages, in this paper we concentrate on *attribute handling* for relational specifications of model transformations for the *special case of TGGs*. Attribute handling is of general interest in the context of model transformation languages allowing for relational specification. Therefore, we assume that the way we propose attribute handling in this paper for the TGG case, may help as a guideline to come to a more relational kind of attribute handling in QVT Relational including a consistent operationalization, too.

First, we present a way of *specifying attribute conditions* for TGGs *in a relational way* by adopting *symbolic attribute handling* [OL10]. In particular, it allows for specifying relations between attribute values of the source and target models in a TGG rule making use of first-order formulas over the corresponding attribute labels. An attribute label uniquely identifies a specific attribute in the left-hand side (LHS) or right-hand side (RHS) of the TGG rule. In order to be able to implement this symbolic approach directly, the integration of a constraint solver into the tool environment is supposed. However, such an integration is not always available or not always appropriate because of reasons that we discuss in Section 4. Therefore, we present the *characteristic barriers* that may occur in each TGG application scenario when trying to *operationalize a relational kind of attribute specification* into *consistent* attribute constraints that can be checked before applying the corresponding operationalized TGG rule as well as into consistent attribute computation instructions for created elements. These attribute constraints and attribute computation instructions can then be evaluated and computed in regular TGG tools in a straightforward way (without having to rely on an integrated constraint solver). Moreover, we present some *pragmatic solutions* (partially implemented already, or being implemented) to overcome the operationalization barriers for different TGG application scenarios in our own TGG implementation. Finally note that the operationalization conditions for relational attribute formulas in each TGG application scenario as proposed in this paper are designed to ensure the preservation of consistency with the TGG. However they do not necessarily guarantee *completeness* in the sense that not each potential consistent attribute assignment is being pursued.

The outline of this paper is as follows. Section 2 reintroduces TGGs with its application scenarios (without attribute handling). Section 3 then introduces attribute formulas for TGG rules using the symbolic graph transformation approach. The following sections describe the barriers one has to overcome when operationalizing these attribute formulas for the different TGG application scenarios: TGG execution (Section 4), TGG integration 5, and TGG forward

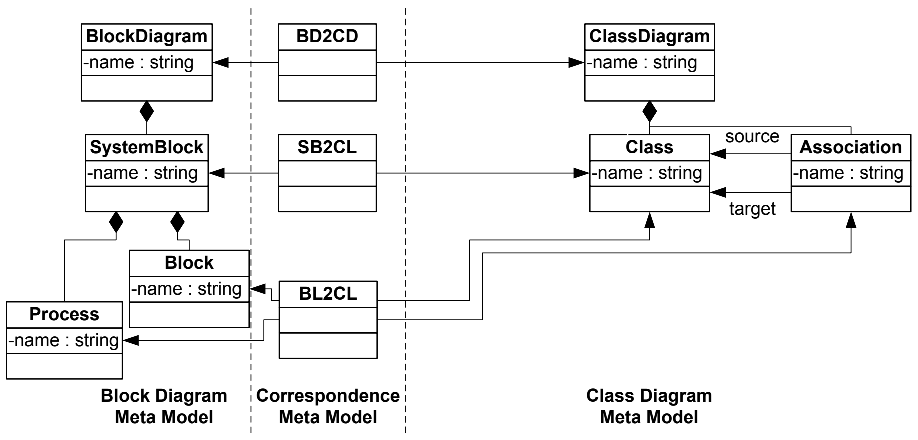


Figure 1: Example metamodels

and backward model transformation 6. Finally, Section 7 discusses related work and in Section 8, we summarize our work and describe how the development of this approach can be continued.

2 Triple Graph Grammars Revisited

As a running example we will use a model transformation[1] from SDL block diagrams [ITU02] to UML class diagrams. The metamodels of these languages are shown in Fig. 1. Block diagrams are hierarchical structures, where a *BlockDiagram* contains *SystemBlocks* which in turn contain *Blocks* and *Processes*. In the class diagram, a *ClassDiagram* contains all other elements. These are *Classes* that can be connected to each other via *Associations*. All these elements can have a name. There is also a correspondence metamodel. Its elements connect elements of the other two metamodels. This way, the correspondence model stores traceability information, which allows to find elements of one model that correspond to elements of the other model.

Triple graph grammars (TGG) describe how to relate source and target models of a model transformation via so-called correspondences. In particular, a TGG consists of an axiom (the grammar's start graph) and several TGG rules over a source, correspondence and target domain. The TGG for the transformation of block and class diagrams is shown in Fig. 2[2]. We use a short notation that combines the left-hand (LHS) and right-hand sides (RHS) of the graph transformation rule. Elements that belong to the LHS and RHS are black, elements that belong only to the RHS (i.e. which are created by the rule) are green and marked with "++". TGG rules are divided into three domains: The source model (left), target model (right), and the correspondence model domain (middle). The axiom in Fig. 2 relates the root elements of the source and target models with the axiom correspondence node[3]. Rule 1 creates a *SystemBlock* and a corresponding *Class*. The *BlockDiagram* and *ClassDiagram* must already exist. Rule 2 creates a *Block* and a *Pro-*

[1] This model transformation is a simplified version of a transformation used in the industrial case study on flexible production control systems [SWGE04].
[2] Note, that the types defined in Fig. 1 are abbreviated in Fig. 2.
[3] These root elements serve as containers for all other model elements to ease processing of the models.

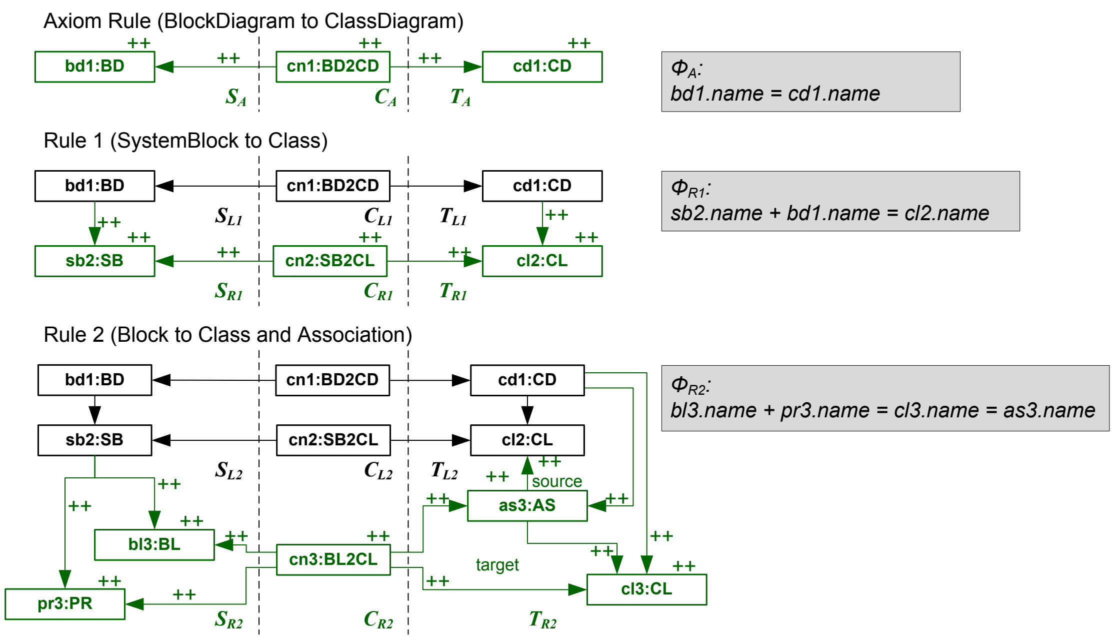


Figure 2: Example TGG rules and axiom rule with relational attribute specification

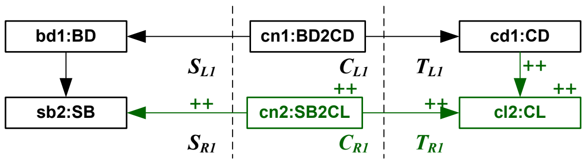


Figure 3: Operational forward rule r_1^F derived from rule r_1

cess in the block diagram domain and connects them to the *SystemBlock*. In the class diagram domain, a class is created and connected to the *SystemBlock*'s *Class* with an *Association*.

TGGs may have *different application scenarios*. The TGG itself can be used to build source and target models connected by correspondences in parallel by *executing the underlying grammar*. This means that TGG rules are applied successively to extend the axiom such that in the resulting graph the source and target components (i.e. the source and target models) are consistent with each other. This application scenario can be useful to automatically generate test cases for a model transformation implementation derived from a TGG as described in [HLG+11].

In the more usual application scenario, either a source or target model already exists and a *model transformation system* has to create the other one according to the given TGG. In fact, TGGs cover *three kinds of model transformation directions*: Forward, backward, and correspondence transformations. A forward (backward) transformation takes a source (target) model as input and creates the correspondence and target (source) model. A correspondence[4] transformation requires a source and target model and creates only the correspondence model. In order

[4] The correspondence transformation is also known as mapping transformation or model integration.

to derive, for example, a forward model transformation, *operational rules* have to be generated from the TGG, which create target model elements for given source model elements, so that both are consistent with each other. In particular, all elements in a TGG rule belonging to the source domain that were previously created are added to the LHS of the rule. A forward rule then specifies how source elements can be translated into target elements according to the TGG. See Fig. 3 depicting the forward rule r_1^F derived from rule r_1 of our running example TGG. The transformation of attribute values is not shown here but will be explained in detail in Sec. 6. For each of the aforementioned transformation directions, separate operational rules are derived. We denote the forward rule of r by r^F, the backward rule of r by r^B (see Fig. 8) and the integration rule of r by r^I (see Fig. 7). In order to ensure consistency with the TGG it has to be ensured that a given source model element is only transformed once by the corresponding operational rules. This requires a bookkeeping mechanism, which keeps track of those elements that were already transformed, and those that still have to be transformed. In [GHL12] we describe how to come from the relational TGG description to a conform and efficient batch transformation implementation covering one of these directions. Note that it is also possible to use the TGG to perform synchronizations of source and target models [GH09, GW09], but this is not within the main concern of this paper.

We have developed an *implementation of TGGs* based on Eclipse and the Eclipse Modeling Framework. The system can perform model transformation and model synchronization as well as automatically generate test cases for the model transformation as described above.[5] It utilizes several optimizations to increase performance of model transformations.

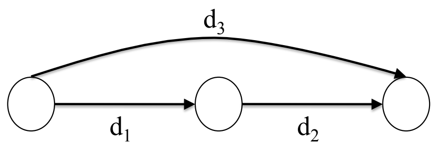
3 Relational Attribute Specification in TGGs

In this section, we present a way of specifying attribute conditions for TGGs in a relational way by adopting symbolic attribute handling. Therefore, we start with a short and informal introduction to symbolic graphs and graph transformation.

Symbolic graphs [OL10] can be seen as a specification of a class of attributed graphs (i.e. of graphs including values from a given data algebra in their nodes or edges). In particular, in a symbolic graph, values are replaced by variables and, moreover, a set of formulas, Φ, specifies the values that the variables may take. Then, we may consider that a symbolic graph $SG = \langle G, \Phi \rangle$ denotes the class of all graphs obtained by replacing the variables in the graph G by values that satisfy Φ. For instance, the symbolic graph in Figure 4 specifies a class of attributed graphs, including distances in the edges, that satisfy the well-known triangle inequality. A symbolic graph thus can be seen as a specification of a class of attributed graphs, where variables in the symbolic graph are attribute labels (usually denoted `objectName.attributeName`), uniquely identifying the attribute of a particular node (or edge), together with a set of formulas Φ over these variables constraining the respective attribute values.

In *symbolic graph transformation* we consider that the LHS and RHS of a rule r are symbolic graphs, where the conditions $\Phi_{L,r}$ on the LHS of r are included in the conditions Φ_r in the

[5] It can be downloaded from our Eclipse update site http://www.mdelab.de/update-site



with $d_3 \leq d_1 + d_2$

Figure 4: A symbolic graph

right hand side of the rule $r : \langle \Phi_{L,r}, L \hookrightarrow R, \Phi_r \rangle$. This means that applying a symbolic rule to a symbolic graph $\langle G, \Phi_G \rangle$ reduces or narrows the instances of the result. For instance, G may include an integer variable x such that Φ_G does not constrain its possible values. However, after applying a given transformation, in the result graph $\langle H, \Phi_H \rangle$ we may have that Φ_H includes the formula $x = 0$, expressing that 0 is the only possible value of x. Concluding, $\Phi_{L,r}$ is a formula expressing how the attribute values in a graph to be transformed should be constrained, and Φ_r expresses in addition how the attribute values of the result graph are to be constrained. Note that only the manipulation of attributes is symbolic, the graph changes performed by a symbolic rule are performed as usual in the double-pushout approach. For a more formal definition of symbolic graph transformation we refer to [OL10].

When *defining triple graph rules using symbolic graph transformation*, attribute values in a rule r are thus constrained in a declarative way by a set of formulas Φ_r expressing their relation with other attribute values of elements in r. Following the symbolic approach [OL10], we assume that Φ_r is a set of first-order formulas over a set $X_r = \{x_1, \ldots, x_n\}$ of free variables, being attribute labels of attributes in the LHS or RHS graph of the rule, and constants belonging to the respective attribute domains. Each TGG rule r is thus equipped with a set of formulas Φ_r, and we say that Φ_r is the so-called *attribute formula*[6] of the TGG rule r. Note that we assume for each TGG rule r that attribute values of preserved elements are not changed, since a grammar usually does not change anything which has been created before. The attribute specification for TGGs in our *TGG implementation* can be done by OCL[7] constraints implementing the attribute formula of each TGG rule.

Example 1 *As shown in Fig. 2, each TGG rule contains an attribute formula. Note that in our examples we use OCL syntax. The free variables in the axiom's formula Φ_A are the attribute labels* bd1.name *and* cd1.name*. The formula states that both must be equal. The formula Φ_{R1} of rule $R1$ states that the* cl2.name *is the concatenation of the names of* sb2 *and* bd1 *and in rule $R2$ the names of* cl3 *and* as3 *must be equal to the concatenation of the names of* bl3 *and* pr3.

Not only the TGG language as relational specification technique is having its difficulties to allow for defining attributes in a declarative way. The following example illustrates that the current solution in QVT Relational[Obj11] for specifying attributes has a rather operational (instead of declarative) flavor. Consequently, the way we propose attribute handling for TGGs in this

[6] For simplicity, we say attribute formula, instead of attribute formulas, since a set of formulas can be joined to one formula using the conjunction operator.
[7] http://www.omg.org/spec/OCL/

paper could help as a guideline to bridge similar gaps for attribute handling in other relational specification techniques.

Example 2 *In Fig. 5, a listing of a QVT Relational rule that we assume to correspond as close as possible to the example TGG rule 1 (from Fig. 2) is shown. Lines 7, 11, and 16 correspond to the attribute formula* Φ_{R1}*. However, instead of expressing this relation directly, the free variables* n1 *and* n2 *have to be used to store the names of the* BlockDiagram *and the* SystemBlock*. Furthermore, the statement in line 19 is required to ensure that* n2 *can be computed in both transformation directions. The QVT standard specifies*[8] *that all free variables have to be bound in expressions of the form* <object>.<property> = <variable> *in the source domain, when, or where clauses. In the forward direction,* n2 *can be bound in line 7 because* sdl *is the source domain in that case. For the backward direction, a statement how to calculate* n2 *must be specified separately in the where clause. This way of specifying attribute conditions and assignments in QVT Relational has a rather operational flavor.*

```
 1  top relation SystemBlock2Class
 2  {
 3    n1 : String;
 4    n2 : String;
 5
 6    enforce domain sdl system : blockDiagram::SystemBlock{
 7      name = n2,
 8      blockDiagram = blockDiagram : blockDiagram::BlockDiagram{}};
 9
10    enforce domain uml umlClass : classDiagram::Class{
11      name = n1 + n2,
12      classDiagram = classDiagram : classDiagram::ClassDiagram{}};
13
14    when {
15      BlockDiagram2ClassDiagram(blockDiagram, classDiagram);
16      blockDiagram.name = n1;}
17
18    where {
19      umlClass.name.substringAfter(n1) = n2;}
20  }
```

Figure 5: QVT Relational transformation rule corresponding to TGG rule 1

Concluding, TGGs can be used to specify model transformations in a declarative (relational) way and symbolic graph transformation can be used to also specify the relation between attribute values in a TGG rule r in a declarative (relational) way. However, when using TGGs in different application scenarios such as forward/backward model transformation, model integration or also TGG execution, attribute values need to be computed that are consistent with the TGG rule formula Φ_r. A possible way to do this would be to *integrate a constraint solver* into the TGG transformation tool able to directly interpret the symbolic approach. It would check at runtime the consistency of attribute values for preserved elements and compute valid values for created elements of the TGG operational rules according to the attribute formula Φ_r of the TGG rule r.

[8] cf. Sec. 7.5 of [Obj11]

In the next section, we describe why this is not always appropriate.

4 Attribute Computation for TGG Execution

A *constraint solver* may be a powerful tool to compute consistent attribute values *at runtime*. However there are certain situations where a different solution may be more adequate. First of all, depending on the kind of constraints considered, solving them may be undecidable or unfeasible. However, this is not the only reason. Let us consider some examples to describe some other problems. Suppose, as it happens in our running example, that the value of a target attribute a^t is defined to be the concatenation of two source attributes a_1^s and a_2^s. Then, in the case of a forward transformation, we may consider that directly computing a^t as the concatenation of a_1^s and a_2^s will be more efficient than using a solver for the same purpose. Conversely, in the case of a backward transformation, we may consider it as reasonable to use a constraint solver to generate all the possible values of a_1^s and a_2^s for a given value of a^t. However, in many cases, it may be more reasonable to assume that the user would input, via input parameters, a specific decomposition of a^t. A similar situation is the case of an information-adding model transformation, where the user of the model transformation might need to set a specific attribute of the target model to a concrete value because the specification does not include any constraints on this attribute. In this case, a constraint solver could generate as possible solutions all the values of the given data domain. Obviously, in this case, it will probably make more sense to assume that the user will input that value. Concluding, because of different reasons it might be more appropriate for a TGG tool not to use an integrated constraint solver at runtime, but to revert to a more pragmatic approach based on specific instructions to check and compute the attributes. In the following sections, we discuss *for each application scenario* how a desirable operationalization of the attribute formula would look like and *which gap needs to be bridged* in this case *between the relational attribute formula and this operationalization*.

In this section, in particular, we are concerned with the TGG execution scenario allowing for automatic test case generation as shown in [HLG+11]. In this application scenario, we need to be able to apply TGG rules non-deterministically[9] to the axiom, have attribute conditions for the TGG rule that can be checked before applying the rule, and set attribute values of created elements using concrete attribute computation instructions that are consistent with the original attribute formula of the TGG rule.

More formally, we need the following *operationalization conditions*: (1) for each attribute label x_j of some created element of a TGG rule r, there exists an *attribute computation instruction* $x_j := t$, where t is a term over constants, input parameters p_k with $1 \leq k \leq m$, or attribute labels x_i with $1 \leq i \leq n$ and $i \neq j$ such that x_i is a variable used as an attribute label of a preserved element of the TGG rule r. Input parameters p_k are free variables in t not used as attribute labels. These computation instructions need to be consistent with the attribute formula Φ_r, i.e. each term t, representing the attribute computation of attribute label x_j of some created element in r describes a *valid solution* for the attribute formula Φ_r of the TGG rule r. (2) When *matching* the TGG rule r, the part $\Phi_{L,r}$ of the attribute formula Φ_r of r constraining values of the LHS of the

[9] Non-determinism arises since more than one rule might be applicable and there might be more than one match available for the same rule.

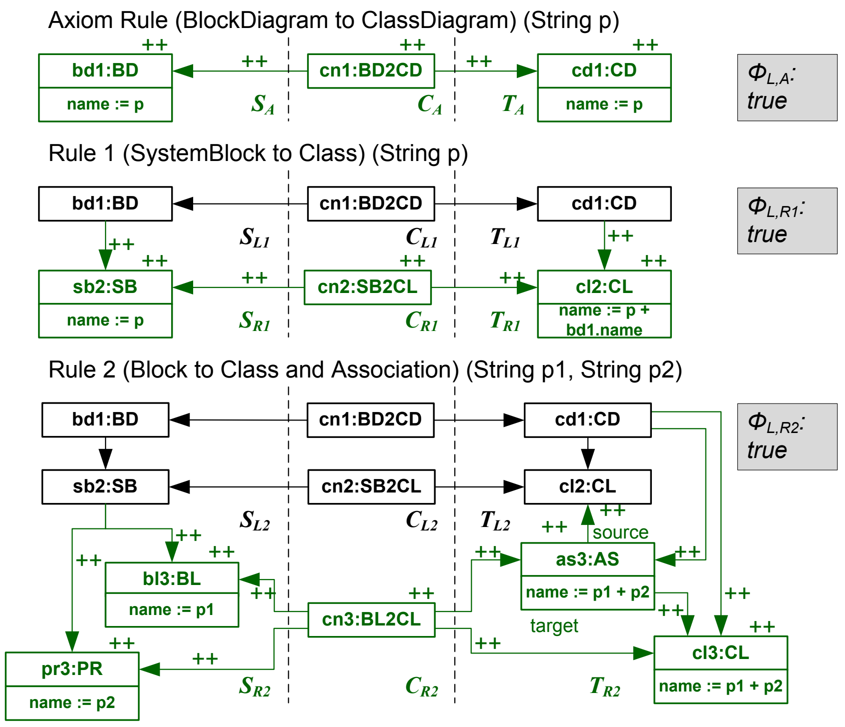


Figure 6: Example TGG rules and axiom rule with computation instructions and checks for execution scenario

rule needs to be *satisfied*.

Example 3 *Fig. 6 shows rules that can be used for the TGG execution scenario without the integration of a constraint solver at runtime. The rules here have input parameters and attribute computation instructions for each attribute of a created element. The attribute formulas do not constrain the values of the LHS of the rule, therefore* $\Phi_{L,r}$ *equals* $true$ *and the* operationalization condition *2 is satisfied in any case. The attribute formula* Φ_A *of the axiom is* bd1.name = cd1.name. *When we execute the TGG like a grammar, both elements are created and no name exists yet. Therefore, we introduce an input parameter* p *for the axiom rule. Accordingly, the attribute computation instructions for the created elements are* bd1.name := p *and* cd1.name := p. *This is similar for rule* $R1$ *but the attribute computation instruction* cl2.name := p + bd1.name *is slightly more complex because it not only contains the parameter* p *but also* bd1.name. *Note that* p + bd1.name *is a term* t *over the input parameter* p *of rule* $R1$ *and over the attribute label* $bd1.name$ *belonging to a preserved element. Moreover, the attribute computation instructions* cl2.name := p + bd1.name *and* sb2.name := p *are consistent with the corresponding attribute formula* sb2.name + bd1.name = cl2.name *in Fig. 2 since they represent valid solutions for this formula. As described in the future work section, deriving valid attribute computation instructions automatically is an open challenge. Rule* $R2$ *even needs two input parameters* p_1, p_2 *for the names of* bl3 *and* pr3, *since then we can transform* Φ_{R2}: bl3.name + pr3.name = cl3.name =

as3.name *into computation instructions for* $bl3.name$,$pr3.name$,$cl2.name$, *and* $as3.name$ *over* p_1 *and* p_2. *In particular, the concatenation of both parameters is used as the name of* cl3 *and* as3.

Our *TGG implementation* enables the TGG developer to input the above-described attribute computation instructions (expressed in OCL) with input parameters (for the moment restricted to type $String$) for the TGG execution scenario. Consistency of the attribute computation instructions (inserted by the developer) with the attribute formula Φ_r can be checked by the tool at runtime after each rule application by evaluating Φ_r.

5 Attribute Computation for TGG Model Integration

In this section, in particular, we are concerned with the TGG model integration scenario. In this application scenario, we need to be able to apply TGG integration rules (creating correspondences) to a source model and target model in order to check if they are consistent with each other with respect to the TGG. When applying a TGG integration rule derived from a TGG specification rule (without the integration of a constraint solver at runtime), we need attribute conditions for the integration rule that can be checked before applying the rule. However the attribute formula Φ_r already is in this form, since correspondence nodes usually do not contain any attributes. As a consequence, no attribute computation instructions need to be available for created elements either.

More formally, we need the following *operationalization condition*: (1) When *matching* the integration rule r^I, the part Φ_{L,r^I} of the attribute formula Φ_r of r constraining values of the LHS of the integration rule r^I, needs to be *satisfied*. Assuming that correspondence elements do not contain attributes and since all source and target elements of a TGG rule r belong to the LHS of the integration rule r^I, in particular $\Phi_r = \Phi_{L,r^I}$, and it is sufficient to check that Φ_r holds.

In the more rare case that *correspondence elements* contain *attributes*, we have an extra operationalization condition in order to ensure that correspondence elements are created with attribute values consistent with the corresponding attribute formula in the TGG: For each attribute label x_j of some created correspondence element of an integration rule r^I, there exists an *attribute computation instruction* $x_j := t$, where t is a term over constants, input parameters p_k with $1 \leq k \leq m$ or attribute labels x_i with $1 \leq i \leq n$ and $i \neq j$ such that x_i is a variable used as an attribute label of a preserved element of the integration rule r^I. Input parameters p_k are free variables in t not used as attribute label. Each term t, representing the attribute computation of attribute label x_j of some created element in r^I describes a *valid solution* for the attribute formula Φ_r of the TGG rule r.

Example 4 *The attribute formulas in Fig. 2 can be taken as they are in order to be checked when applying the integration rule (see Fig. 7), since they constrain only attributes of the LHS of the integration rule. For example, the attribute labels constrained by the attribute formula* Φ_{R1} *of TGG rule* $R1$ *belong to the LHS of the corresponding integration rule* $R1^I$ *and, therefore,* $\Phi_{R1} = \Phi_{L,R1^I}$.

Our *TGG implementation* assumes that correspondence nodes do not possess attributes. Therefore, when matching an integration rule r^I, it is enough to check if $\Phi_r = \Phi_{L,r^I}$ is fulfilled.

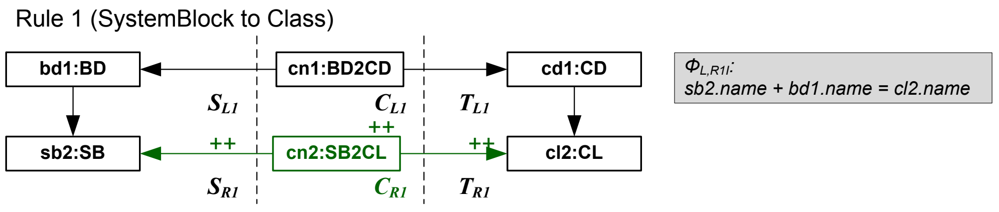


Figure 7: Integration rule $R1^I$

6 Attribute Computation for TGG Forward or Backward Model Transformation

In this section, in particular, we are concerned with the TGG model transformation scenario (forward or backward). In this application scenario, we need to be able to apply forward TGG rules to a source model in order to obtain a consistent target model. When applying a TGG forward rule derived from a TGG specification rule (without the integration of a constraint solver at runtime), we need attribute conditions for the forward rule that can be checked before applying the rule and set attribute values of created elements using concrete attribute computation instructions that are consistent with the original attribute formula of the TGG rule.

More formally, we need the following *operationalization conditions* in the forward case: (1) For each attribute label x_j of some created element of a forward rule r^F, there exists an *attribute computation instruction* $x_j := t$, where t is a term over constants, input parameters p_k with $1 \leq k \leq m$ or attribute labels x_i with $1 \leq i \leq n$ and $i \neq j$ such that x_i is a variable used as an attribute label of a preserved element of the forward rule r^F. Input parameters p_k are free variables in t not used as attribute labels. Each term t, representing the attribute computation of attribute label x_j of some created element in r^F describes a *valid solution* for the attribute formula Φ_r of the TGG rule r. (2) When *matching* the forward rule r^F, the part Φ_{L,r^F} of the attribute formula Φ_r of r constraining values of the LHS of the forward rule r^F needs to be *satisfied*.

Example 5 *Fig. 8 shows rules, which can be used for the TGG backward model transformation scenario without the integration of a constraint solver at runtime. The attribute computation in the axiom can be derived directly from the attribute formula. Additional constraints on attribute values of LHS elements do not exist. In rule* $R1$*, the attribute formula* Φ_{R1} *is* sb2.name + bd1.name = cl2.name. *Therefore, a valid computation instruction for the name of the created element* sb2 *is the name of* cl2 *without the name of* bd1*, since* sb2.name := cl2.name - bd1.name[10] *is a valid solution of* Φ_{R1} *and the term* cl2.name - bd1.name *consists of attribute labels of preserved elements. But the attribute formula* Φ_{R1} *also implies a non-trivial LHS attribute condition* $\Phi_{L,R1^b}$*: In a valid match, the name of* cl2 *must start with the name of* bd1*. As described in the future work section, deriving these LHS conditions automatically is an open challenge. In rule* $R2^b$*, the names of* bl3 *and* pr3 *have to computed. The attribute formula of* $R2$ *specifies that the*

[10] For simplicity reasons, we introduced a string operation $-$ that performs the described subtraction of strings. If a corresponding *BlockDiagram* to the *ClassDiagram* under translation according to the TGG exists, then this partial operation is also well-defined.

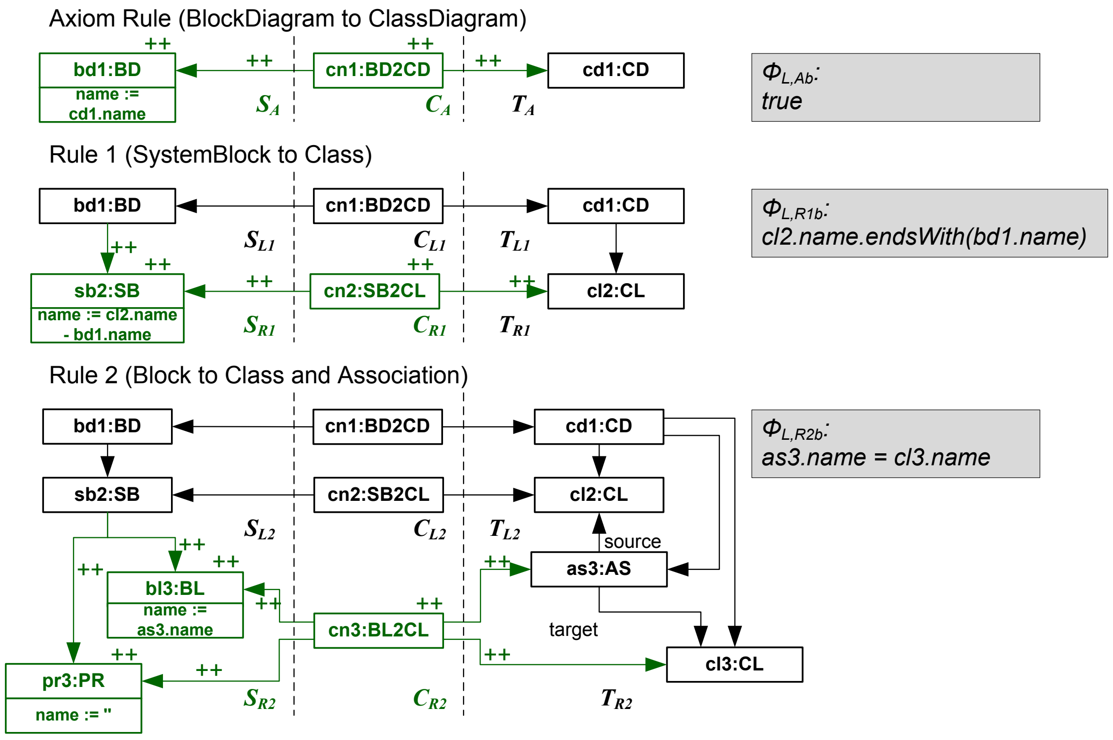


Figure 8: Example TGG rules and axiom rule with computation instructions and checks for backward direction

concatenation of both names is equal to the name of as3 *and* cl3*, which implies for the LHS attribute condition* $\Phi_{L,R2^b}$ *that* as3.name = cl3.name*. There are multiple possibilities to split the attribute formula of $R2$ into valid attribute instructions for the created elements* bl3 *and* pr3*. In Fig. 8, it was decided to take the complete name of* as3 *for the name of* bl3 *and set the name of* pr3 *to the constant empty string. Other possibilities would be to do it the other way round and set* bl3 *to the empty string or split the name of* as3 *in the middle. The example rules in Fig. 8 do not contain any input parameters. In case of rule $R2$, an integer parameter could be introduced that specifies the index, where the name of* as3 *must be split. But then we would have to handle the case that the length of* as3.name *is less than this index.*

Example 6 *Suppose that the example TGG rules now work with a target metamodel, where each Class does not only possess the attribute name, but also the attribute $attr$. The relational attribute formula of each TGG rule over this new target metamodel does not change, since this information is only available in the target model for the software engineer. Therefore, the value of attribute $attr$ is not constrained and the user of the forward model transformation could be allowed to set the attribute to a concrete value. This means that $cl2.attr := x$ with x an input parameter for the $attr$ domain would be a valid solution of the attribute formula of $R1$. The user can choose to set the value of x after the transformation, information is thus added to the target model and forgotten again when performing the backward transformation.*

Our *TGG implementation* enables the TGG developer to input the above-described attribute computation instructions as well as LHS attribute conditions for the forward and backward direction. Consistency of these instructions (inserted by the developer) and conditions with the attribute formula Φ_r can be checked at runtime after each rule application by evaluating Φ_r.

Note that in many application scenarios, one expects the model transformation specification to describe model transformations with *functional behavior*, i.e. a bijective mapping between source and target models. In [HEGO10, GHL12] it is described how to check this statically relying on critical pair analysis. Having also attribute formulas on the rules, they should be taken into account and integrated into an extended critical pair analysis. This would involve constraint solving of the attribute formulas in order to find out if multiple solutions (as sketched in Example 5) for a particular transformation direction exist. It is part of future work to describe this issue in detail and investigate its relation to reversible programming languages [YAG08].

7 Related Work

Attribute handling for TGGs has been discussed already in [KW07], where the gap between a relational specification of attributes in the form of constraints (similar to the symbolic approach proposed in this paper) and the corresponding potentially inconsistent operationalization for different TGG application scenarios in the form of assignments has been identified. A detailed and formal discussion of the barriers that need to be overcome to bridge this gap is not given, but picked up in this paper. In order to keep the gap small, some tools allow only restricted attribute constraints for which the consistent forward and backward operationalization into computation instructions for attribute assignments is straightforward to realize. For example, in [GK10] so-called attribute equality constraints are introduced that allow for specifying which source and target element attributes should be equal. As future work, it is described that it would be desirable to be able to integrate, for example, the Object Constraint Language (OCL) to express attribute value constraints. Also in [SK08], it is mentioned that in terms of enhancing expressiveness of the TGG approach, the handling of complex attribute conditions should be available. In [EEE$^+$07] attributes for TGGs are supported and the running example makes the impression as if relations between attribute values on the source and target side need to be specified using assignments with common variables. However, there is no guideline about how to operationalize these attribute expressions and which limitations thereby might hold in order to achieve consistency in all TGG application scenarios.

It is also possible to circumnavigate the above described gap between relational attribute specification and its operationalization in the form of concrete attribute computation instructions and attribute condition checks. In [CCGL10, GLO09], for example, OCL attribute conditions are added to the TGG rules and their operationalization is avoided by the proposal to use a constraint solver which is integrated into the tool environment guessing the right attribute values fulfilling the attribute conditions at runtime. The authors admit however that relying purely on constraint solving at the operational level may present computational efficiency problems in some cases. Note that [GLO09, OGLE09] proposes a more flexible kind of declarative (pattern-based) specification for model transformations as TGGs, but analog issues occur when trying to handle attribute specification and computation.

Using TGGs to specify transformations between metamodels with big structural and/or semantic differences can be quite problematic. This is a general problem of relational model transformation approaches and not in the focus of this paper. Enhancing the expressiveness of TGGs is subject to current research. For example, application conditions can be used to enhance expressiveness as described in [KLKS10, HEGO10]. Also [GK10] is concerned with expressiveness of TGGs and compares them to the QVT standard [Obj11]. Note that adding new features to TGGs enhancing their expressiveness makes the task of proving conformance more complex. With respect to attribute handling, QVT Relational proposes some restrictions on attribute expressions that should ensure operationalization of the model transformation specification. This specification is quite restrictive and not purely declarative, meaning that concrete attribute computation instructions need to be given for both transformation directions (see Example 2 for more details). Consistency of both transformation directions with each other is not discussed.

8 Conclusion and Future Work

We presented an approach being able to specify bidirectional transformations with TGGs in a declarative (relational) form also on the attribute level. Moreover, we described for each TGG application scenario how the operational form of attribute computation should look like in order to be able to perform test case generation, forward/backward model transformation and model integration without the integration of a constraint solver into the TGG tool environment at runtime. This characterization clarifies the *barriers* that TGG implementation developers need to take when *operationalizing relational (declarative) attribute specifications*. We sketched how our own TGG implementation overcomes these barriers in a pragmatic way by allowing the TGG developer to input computation instructions and LHS conditions for each application scenario as well as by consistency checks at runtime.

Of course, it would be more appropriate for the TGG developer to *simply input the relational (declarative) attribute specification* for the TGGs once, and not having to input left-hand side rule attribute conditions and computation instructions for each application scenario. Investigating how to compute these automatically at design time is part of future work. An intermediate step could be to require as input the left-hand side conditions and instructions from the TGG developer, but check automatically at design time that they are consistent with the relational formula. Alternatively, a more pragmatic solution for this problem would be to create and maintain a library of relational attribute formulas for TGGs with their operationalizations in the form of computation instructions and LHS rule attribute conditions for the different application scenarios. TGG tools could then support a relational input of attribute specifications according to this library. As mentioned earlier, the operationalization conditions proposed in this paper guarantee merely the preservation of consistency with the TGG, but not *completeness*. It is part of future work to investigate how to also cope in an appropriate way with different consistent attribute formula operationalizations. It is part of future work to describe this issue in detail and investigate its *relation to reversible programming languages* [YAG08] as well as to similar properties e.g. for schema mappings in the *database literature*. Finally, it should be investigated in which TGG application scenarios *lazy evaluation* of attributes [OL12] could be convenient.

References

[CCGL10] J. Cabot, R. Clarisó, E. Guerra, J. Lara. Verification and validation of declarative model-to-model transformations through invariants. *J. Syst. Softw.* 83(2):283–302, 2010.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In Dwyer and Lopes (eds.), *Fundamental Approaches to Software Engineering*. LNCS 4422, pp. 72–86. Springer, 2007.

[GGL+06] H. Giese, S. Glesner, J. Leitner, W. Schäfer, R. Wagner. Towards Verified Model Transformations. In Hearnden et al. (eds.), *Proc. of the 3^rd International Workshop on Model Development, Validation and Verification (MoDeVa), Genova, Italy*. Pp. 78–93. Le Commissariat à l'Energie Atomique - CEA, October 2006.

[GH09] H. Giese, S. Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical report 28, Hasso Plattner Institute at the University of Potsdam, 2009.

[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.

[GK10] J. Greenyer, E. Kindler. Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. *Software and Systems Modeling* 9(1):21–46, 2010.

[GLO09] E. Guerra, J. de Lara, F. Orejas. Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions. In Paige (ed.), *ICMT*. Lecture Notes in Computer Science 5563, pp. 83–99. Springer, 2009.

[GW09] H. Giese, R. Wagner. From Model Transformation to Incremental Bidirectional Model Synchronization. *Software and Systems Modeling (SoSyM)* 8(1), 2009.

[HEGO10] F. Hermann, H. Ehrig, U. Golas, F. Orejas. Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In *Proceedings of the First International Workshop on Model-Driven Interoperability*. MDI '10, pp. 22–31. ACM, New York, NY, USA, 2010.

[HLG+11] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick, I. Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In Schürr and Várro (eds.), *Proceedings of Fourth International Symposium of Application of Graph Transformation with Industrial Relevance (AGTIVE'11)*. 2011.

[ITU02] I. International Telecommunication Union. ITU-T Recommendation Z.100: Specification and Description Language (SDL). 2002.

[KLKS10] F. Klar, M. Lauder, A. Königs, A. Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Engels et al. (eds.), *Graph Transformations and Model-Driven Engineering*. Lecture Notes in Computer Science 5765, pp. 141–174. Springer Berlin / Heidelberg, 2010.

[KW07] E. Kindler, R. Wagner. Triple Graph Grammars: Concept, Extensions, Implementations and Application Scenarios. Technical report, Software Engineering Group, Department of Computer Science, Universitat Paderborn, 2007.

[Obj11] Object Management Group. MOF 2.0 QVT 1.1 Specification. 2011.

[OGLE09] F. Orejas, E. Guerra, J. de Lara, H. Ehrig. Correctness, Completeness and Termination of Pattern-Based Model-to-Model Transformation. In Kurz et al. (eds.), *CALCO*. LNCS 5728, pp. 383–397. Springer, 2009.

[OL10] F. Orejas, L. Lambers. Symbolic Attributed Graphs for Attributed Graph Transformation. In *Graph and Model Transformation 2010*. Volume 30. EC-EASST, 0 2010.

[OL12] F. Orejas, L. Lambers. Lazy Graph Transformation. *Fundamenta Informaticae*, 2012. to appear.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr et al. (eds.), *Proc. of the* $20^{t}h$ *International Workshop on Graph-Theoretic Concepts in Computer Science*. LNCS 903, pp. 151–163. Spinger Verlag, Herrsching, Germany, June 1994.

[SK03] S. Sendall, W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, pp. 42–45, 2003.

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars : Research Challenges, New Contributions, Open Problems. In *4th International Conference of Graph Transformation, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008*. LNCS 5214, pp. 411–425. Springer, Berlin / Heidelberg, 2008.

[Ste10] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and Systems Modeling* 9(1):7–20, 2010.

[Ste11] P. Stevens. A simple game-theoretic approach to checkonly QVT Relations. *Software and Systems Modeling*, pp. 1–25, 2011. Online First.

[SWGE04] W. Schäfer, R. Wagner, J. Gausemeier, R. Eckes. An Engineer's Workstation to support Integrated Development of Flexible Production Control Systems. In Ehrig et al. (eds.), *Integration of Software Specification Techniques for Applications in Engineering*. LNCS 3147, pp. 48–68. Springer Verlag, 2004.

[YAG08] T. Yokoyama, H. B. Axelsen, R. Glück. Principles of a reversible programming language. In *Proceedings of the 5th conference on Computing frontiers*. CF '08, pp. 43–54. ACM, New York, NY, USA, 2008.