Complete Specification Coverage in Automatically Generated Conformance Test Cases for TGG Implementations

Stephan Hildebrandt, Leen Lambers, and Holger Giese

Hasso Plattner Institute, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{stephan.hildebrandt,leen.lambers,holger.giese}@hpi.uni-potsdam.de

Abstract. Model transformations can be specified using an operational or a relational approach. For a relational approach, an operationalization must be derived from the transformation specification using approved formal concepts, so that the operationalization conforms to the specification. A conforming operationalization transforms a source model S to a target model T, which is moreover related to S according to the relational transformation specification. The conformance of an operationalization with its relational specification must be tested since it is not certain that the formal concepts have been correctly realized by the implementation. Moreover, transformation implementations often perform optimizations, which may violate conformance.

The Triple Graph Grammar (TGG) approach is an important representative of relational model transformations. This paper presents an extension of an existing automatic conformance testing framework for TGG implementations. This testing framework exploits the grammar character of TGGs to automatically generate test input models together with their expected result so that a complete oracle is obtained. The extension uses dependencies implicitly present in a TGG to generate minimal test cases covering all rules and dependencies in the TGG specification if the TGG is well-formed. In comparison to the previous random approach, this guided approach allows more efficient generation of higher quality test cases and, therefore, more thorough conformance testing of TGG implementations. The approach is evaluated using several TGGs, including one stemming from an industrial case study.

1 Introduction

Model transformations are an important part of every MDE approach. Therefore, their correctness has to be guaranteed. In a relational model transformation approach, errors may arise from faulty operationalizations, i.e. operationalizations that do not conform to the transformation specification. *Conformance* means that a source model S, which is transformed to a target model T by a transformation implementation, is also related to T according to the relational specification (and vice versa if the specification is bidirectional).

The *Triple Graph Grammar* [14] (TGG) approach is an important representative of *relational model transformation approaches*. To a certain extent, the conformance of a TGG and a corresponding operationalization can be proven by formal reasoning [14,9,5]. In general, though, it is not certain whether implementations have realized each formal concept describing a conforming operationalization correctly. Moreover, usually, TGG formalizations neither cover every technicality that TGG implementations rely on, nor cover each additional optimization that augments the efficiency of the model transformation execution. Therefore, conformance testing of the implementation is required.

A framework for *automatic conformance testing* of TGG implementations was already presented [10], which automatically generates and executes test cases. A test case consists of a source (test input) and an expected target model (test oracle). The testing framework generates random test cases, executes the TGG implementation under test to transform the source model, and compares the created target model with the expected target model. If a difference is detected, a conformance error has been found. The framework's test case generation approach makes use of the *grammar* character of TGGs. TGG rules are randomly applied to create a source and expected target model simultaneously.

To assess the quality of a test case, the framework measures *specification coverage*, which consists of *rule coverage* and *rule dependency coverage.* Rule coverage is the percentage of TGG rules that were applied when building a particular test case. Likewise, rule dependency coverage is the percentage of covered *produce-use* dependencies between TGG rules. The aim of generating test cases is to achieve complete specification coverage. However, when evaluating the random generation approach [10] with several TGGs, complete specification coverage could not be achieved for complex TGGs. If a TGG contains very complex rules, the random generation approach is unlikely to generate test cases covering such rules.

Therefore, this paper presents a different test case generation approach, which generates test cases *guided by dependencies* between TGG rules. In practice, these test cases also achieve complete specification coverage for complex TGGs. Moreover, the test cases are as small as possible, which helps in finding the cause of conformance errors.

This paper is structured as follows: First, Sec. 2 presents the basic principles of TGGs and a running example. Sec. 3 briefly describes the existing conformance testing framework. The new dependency-guided generation approach and its completeness and minimality properties are explained in Sec. 4. An evaluation of the approach follows in Sec. 5, related work is discussed in Sec. 6 and Sec. 7 concludes the paper.

2 Triple Graph Grammars in a Nutshell

Triple Graph Grammars are a relational approach to bidirectional model transformation and model synchronization [14]. TGGs combine three conventional graph grammars for the source, target and correspondence models. The correspondence model explicitly stores correspondence relationships between source and target model elements. Fig. 1 shows the metamodels of *Block Diagrams*,

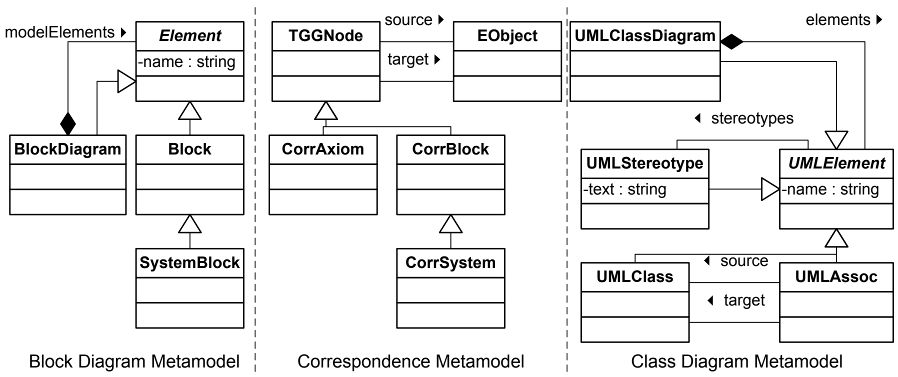

Fig. 1. Example Metamodels

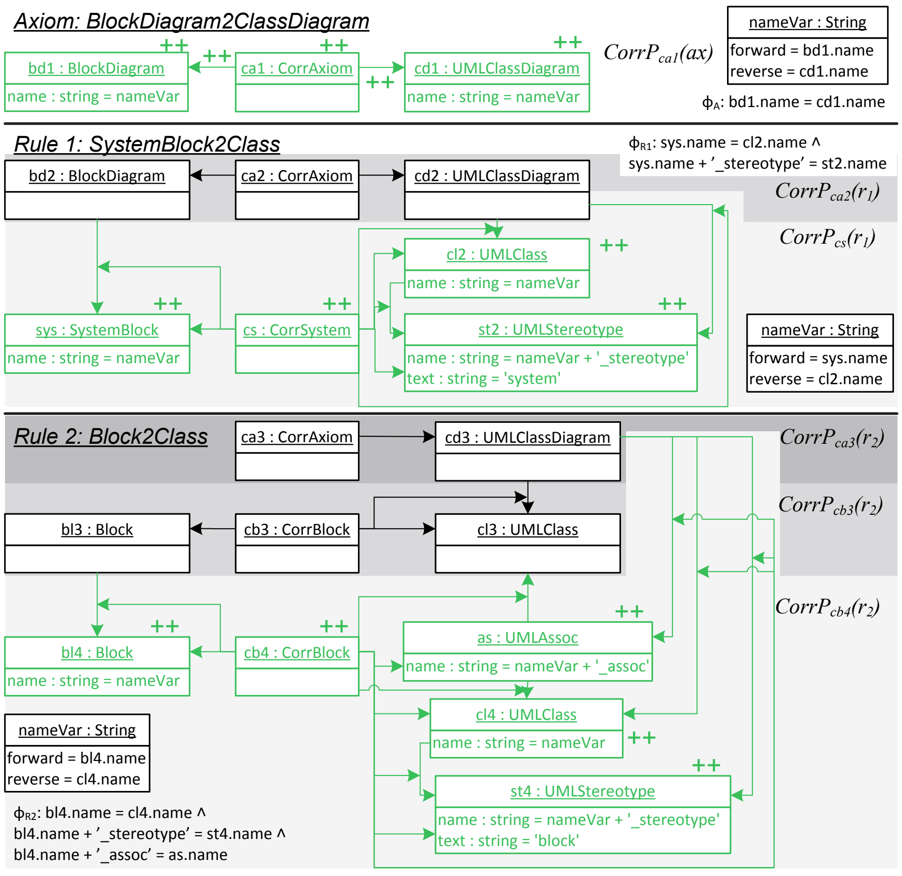

Fig. 2. Example TGG relating Block Diagrams to Class Diagrams

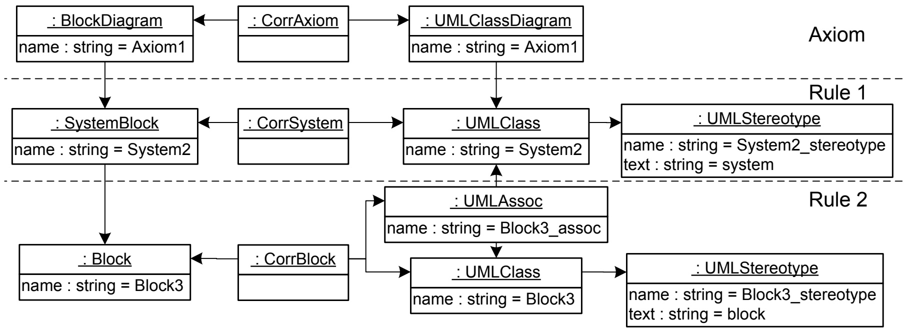

Fig. 3. A Block Diagram and Class Diagram connected by a correspondence model

Class Diagrams and a correspondence metamodel. Fig. 2 shows a TGG specifying a transformation between these languages.

A TGG consists of an *axiom* and several *rules*. Fig. 2 uses a shortened notation, which combines the Left-Hand-Side (LHS) and the Right-Hand-Side (RHS) of a rule. Elements occurring on both sides are black, elements occurring only on the RHS, i.e. which are created by the rule, are green and marked with *++*[1]. TGG rules never delete elements, therefore, the LHS is always a subset of the RHS. In addition, *attribute formulae* (ϕ_i) are specified to ensure consistency of attribute values.

The axiom in Fig. 2 transforms a *BlockDiagram* element to a *UMLClassDiagram* and a *CorrAxiom* node. Rule 1 transforms a *SystemBlock* to a *UMLClass* and a *UMLStereotype* with the *text* “system”. Rule 2 transforms a *Block* to a *UMLAssoc*, *UMLClass* and a *UMLStereotype* with the *text* “block”. Attribute formulae ensure equality of element names. Fig. 3 depicts instances of the metamodels resulting from the following *rule sequence*: Axiom, rule 1, rule 2.

A TGG rule can be applied on a host graph if there is an *injective morphism* from the rule's LHS to the host graph. In practice, type inheritance of the node types must also be respected, i.e. the matched nodes in the host graph must have the same type or a subtype of the node types in the rule. A graph morphism respecting type inheritance is formally defined in [7]. In addition, the rule's attribute formulae must hold. A TGG *rule sequence* is a sequence of the axiom and an arbitrary number of TGG rules. Each rule may appear multiple times in a sequence. A *rule sequence* is *applicable* on a host graph if one rule after the other is applicable starting with the host graph. This implies that all rules in the sequence must only use elements in their LHSs that are available in the initial host graph or are produced by previous rules. If the host graph is empty (as is the case when using the TGG to generate models, cf. Sec. 3), an applicable rule sequence must start with the axiom.

TGGs are relational model transformation specifications that cannot be executed directly to transform a given source model to a target model. Instead, operational rules have to be derived for each transformation direction: A *forward/*

[1] For better readability, only nodes in Fig. 2 are marked with *++*.

backward transformation takes a source/target model (left/right domain in Fig. 2) and creates the correspondence and target/source models. A *model integration* creates the correspondence model for given source and target models.

MoTE[2] is a TGG implementation supporting bidirectional model transformation and synchronization. Since the automatic operationalization of attribute formulae is difficult[13] in general, the developer has to explicitly specify attribute computations for each direction in MoTE. These computations must be compatible with the attribute formulae. When TGG rules are applied directly, as the conformance testing framework does (cf. Sec. 3), attribute values of created elements have to be provided via *rule parameters*, *nameVar* in Fig. 2. For ordinary model transformations, *forward* and *backward* expressions are specified, which compute a parameter's value based on the respective input model.

MoTE's algorithm has been formalized [6] and suitable criteria have been defined[3], which a TGG must satisfy so that MoTE can efficiently execute the transformation and so that conformance is not lost. In addition, these criteria play a crucial role in the *random* and *dependency-guided* test case generation approaches (cf. Sec. 3 and Sec. 4). Among these criteria, the following subset is especially important for the remainder of this paper:

1. Every TGG rule and the axiom create exactly one correspondence node.
2. Every TGG rule contains at least one correspondence node in its LHS.
3. Every model element in a TGG rule (a node or a link in the rule's source or target model domain) is connected to exactly one correspondence node via one correspondence link.
4. Every TGG rule and the axiom always create correspondence links along with their incident nodes.

These criteria have several implications: A single correspondence node in a TGG rule, its outgoing correspondence links and the correspondence links' model elements always form a pattern (criteria 3 and 4). The correspondence node can be used as a representative of that pattern. This is referred to as a *correspondence pattern.* It is denoted as $CorrP_c\,(r)$ when referring to the correspondence pattern of correspondence node c in rule r. Moreover, every rule and the axiom create exactly one correspondence pattern (criterion 1) and every rule contains at least one correspondence pattern in its LHS (criterion 2). Rule 2 (cf. Fig. 2) consists of three correspondence patterns: $CorrP_{ca3}(r_2)$, $CorrP_{cb3}(r_2)$ and $CorrP_{cb4}(r_2)$.

Furthermore, in an *applicable* TGG rule sequence, all correspondence patterns used in the LHS of a rule must be produced by previous rules. In addition to the aforementioned criteria and for the remainder of this paper, all TGGs are assumed to be well-formed according to the following *well-formedness* criterion:

Definition 1 (Well-Formed TGG). *A TGG is* well-formed *if each of its rules satisfies criteria 1 to 4 and an applicable rule sequence exists, which contains that rule.*

[2] `http://www.mdelab.de/mote/`

[3] Note, that MoTE has been developed further since [6] was published. In particular, link bookkeeping has been implemented. Therefore, all criteria demanding to always treat a transformed link along with a transformed node can be relaxed.

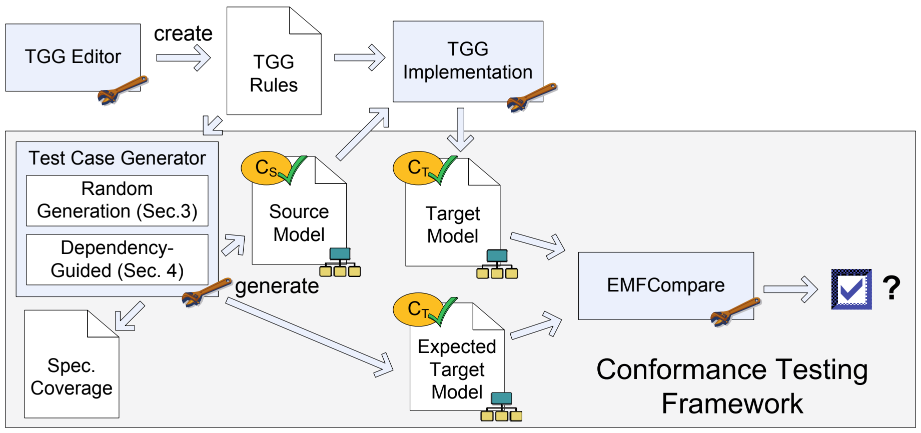

Fig. 4. Components of the automatic conformance testing framework for TGG implementations

A TGG is not well-formed if, for example, one of its rules uses elements that are not produced by any other rule. Such a TGG rule would obviously be unnecessary, comparable to unreachable code in a program. Test cases for such a TGG could never achieve complete *specification coverage.*

3 Automatic Conformance Testing with Random Model Generation

As explained in Sec. 1, a framework for *automatic conformance testing* of TGG implementations was already presented [10]. It is briefly explained in this section.

The testing framework is depicted in Fig. 4. The *Test Case Generator* generates pairs of a source and an expected target model, based on the TGG. This pair forms a test case. It can either use the existing *random* generation approach or the new *dependency-guided* generation presented in Sec. 4. The *TGG Implementation* under test transforms the source model to a second target model, which *EMFCompare* compares to the expected target model. This kind of comparison limits the framework to deterministic TGGs, i.e. there is only one target model per source model.

The existing *random* approach generates random applicable TGG rule sequences and applies them on the empty graph as follows: First, the TGG's axiom is applied to create the root nodes of the three models. The first correspondence node is put into a set. After that, a TGG rule is selected randomly. A match must be provided for each of the rule's LHS correspondence nodes (cf. criterion 2), and, therefore, for all required correspondence patterns. The nodes are selected randomly from the set of previously created correspondence nodes. Then, an attempt is made to apply the rule to extend all three models simultaneously. If this is successful, its created correspondence node is added to the set of previously created correspondence nodes (cf. criterion 1). If the attempt is not successful, another TGG rule is selected randomly and attempted to be applied.

This process is repeated until a user-defined number of rules are applied, which roughly corresponds to the sizes of the generated models. In addition to the desired model sizes, the user has to specify how values of rule parameters should be computed. For each rule parameter, the user can specify whether it should get a fixed value, the value of a counter, or a concatenation of both. In Fig. 3 the values of the *nameVar* rule parameters consist of a fixed string, which corresponds to the name of the rules, and a counter value. The use of rule names for parameter values allows easy retracing of which model element was created by which rule in which order.

Furthermore, the *Test Case Generator* computes *specification coverage*, which consists of *rule coverage* and *rule dependency coverage*. Rule coverage is the percentage of TGG rules that were applied when building a particular test case and rule dependency coverage is the percentage of covered *produce-use* dependencies between TGG rules. In general, a dependency exists between two rules if one rule uses elements in its LHS that are produced by the other rule. Of course, the coverage of a set of test cases should be as high as possible to ensure confidence in the quality of the tested subject. Theoretically, the random approach can achieve complete specification coverage for well-formed TGGs because all existing rule sequences (up to the predefined size) can be generated. However, complex TGG rules may appear only in a small fraction of all possible rule sequences. Therefore, generating such sequences and achieving complete coverage for complex TGGs in practice is unlikely.

Another drawback of the random generation approach is that the test models may become much larger than is actually necessary in order to achieve high coverage. This complicates debugging if a conformance error is found.

4 Dependency-Guided Test Case Generation

To achieve complete *specification coverage*, in particular *rule dependency coverage*, test cases have to be generated to specifically target dependencies present in a TGG. The presented approach analyzes the TGG and makes all dependencies explicit as *rule dependency graphs* (Sec. 4.1). Based on these graphs, *test case descriptions*, which are basically TGG rule sequences, are generated and executed to yield a test case (Sec. 4.2). These test cases achieve complete *specification coverage* and are *minimal* (Sec. 4.3).

4.1 Deriving Rule Dependencies from TGG Rules

The relevant dependencies are *produce-use* dependencies[4] [12]. Therefore, "dependency" will be used synonymously with this term in the remainder of this paper. According to the common definition of *produce-use* dependencies [12], a *produce-use* dependency exists if a rule produces an element that is used by another rule. Due to the criteria imposed on TGG rules (Sec. 2), a *produce-use* dependency between TGG rules can be defined as follows:

[4] Other kinds of dependencies, e.g. delete-forbid dependencies, do not occur because TGGs as presented in Sec. 2 do not delete any elements.

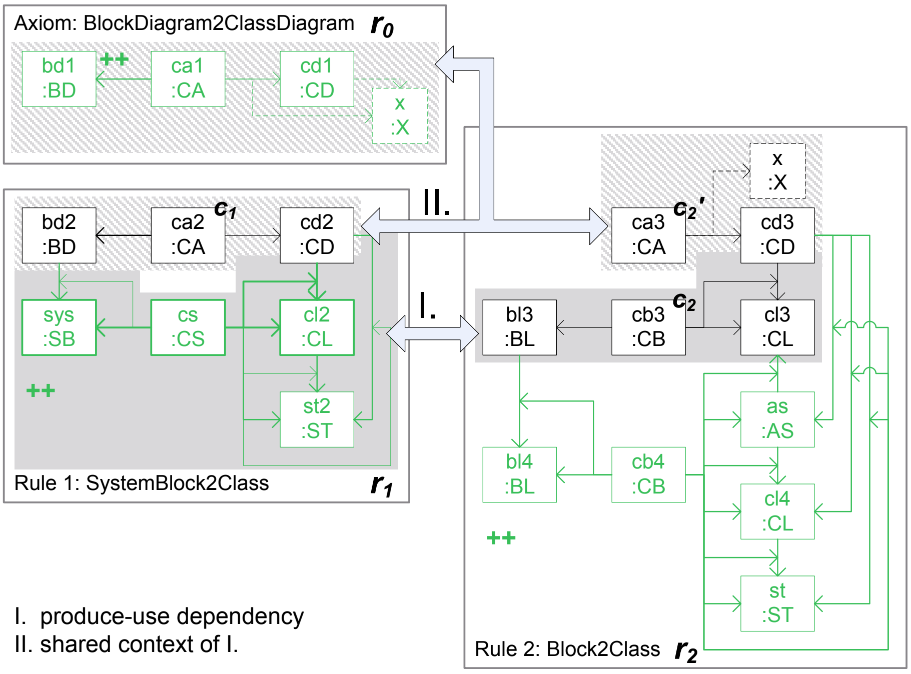

Fig. 5. Dependencies between TGG rules. Node types are omitted. The annotations in italics match the variables used in Definition 2 and Definition 3. Correspondence patterns with same backgrounds can be matched to each other.

Definition 2 (Produce-Use Dependencies between TGG Rules). *A* produce-use *dependency exists from a* required *TGG rule r_1 to a* dependent *rule r_2, and in particular to a correspondence node c_2 in the LHS of r_2, if there is an injective morphism respecting type inheritance between the* correspondence pattern *of c_2 and the correspondence pattern created by r_1. It is denoted $r_1 \rightarrow r_2^{c_2}$.*

Definition 2 is more specific than the general definition of *produce-use* dependencies, because a dependency only exists if the complete correspondence pattern on the LHS of one TGG rule is created by another rule. The classical definition of *produce-use* dependencies also considers the case in which only particular elements of the patterns are used, which would result in a large number of rule dependencies. However, due to criteria 2 and 3 a correspondence pattern is always created by a single rule so that Definition 2 filters out many, but not all (see Sec. 5), dependencies resulting in non-applicable rule sequences. Note, that a TGG rule may have a dependency to itself, e.g. in rule 2, the pattern $CorrP_{cb3}(r_2)$ matches $CorrP_{cb4}(r_2)$ (Fig. 2).

Every TGG rule is applicable in a certain *context*. The *context* of a rule r_2 is a set of TGG rules (or the axiom), which contains one required TGG rule (or axiom) r_i for each correspondence node c_j in the LHS of r_2 so that $r_i \rightarrow r_2^{c_j}$. A TGG rule may be applicable in multiple contexts. The context of rule 1 (cf.

Fig. 5, ignore dashed elements) is the axiom, the contexts of rule 2 are the axiom and rule 1, as well as the axiom and rule 2.

If a rule has multiple correspondence nodes on its LHS, the rule's context can overlap with the context of one of its required rules, i.e. both rules depend on some common rule. The pattern $CorrP_{ca3}(r_2)$ in rule 2 is also present in rule 1 as $CorrP_{ca2}(r_1)$ (dashed backgrounds). Moreover, the combination of the patterns $CorrP_{ca3}(r_2)$ and $CorrP_{cb3}(r_2)$ (grey background) in rule 2 can be found in rule 1. Therefore, rule 2 can only be applied in the context of the axiom and rule 1 if $CorrP_{ca2}(r_1)$ and $CorrP_{ca3}(r_2)$ are matched to the same instance elements when rule 1 and rule 2 are applied. If rule 2 matches $CorrP_{ca3}(r_2)$ to different elements than $CorrP_{ca2}(r_1)$, it is not applicable. The class diagram matched to *cd3* would then not be the class diagram, to which rule 1 added the class *cl2*. The link between *cd3* and *cl3* would not be found. This leads to the definition of a *shared context of a produce-use dependency.*

Definition 3 (Shared Context of a Produce-Use Dependency). *Given a dependency $r_1 \to r_2^{c_2}$ according to Definition 2, a third rule r_0 is a* shared context *of this dependency if the following conditions are satisfied: (1) r_2 contains another correspondence node c_2' in its LHS, $c_2 \neq c_2'$; (2) a dependency $r_0 \to r_2^{c_2'}$ exists; (3) a dependency $r_0 \to r_1^{c_1}$ exists, where c_1 belongs to the LHS of r_1; and (4) there is an injective morphism respecting type inheritance*[5] *from the* restricted *correspondence pattern $CorrPRes_{c_2' \leftrightarrow c_2}(r_2)$ to $CorrP_{c_1}(r_1)$. The* restricted *correspondence pattern $CorrPRes_{c_2' \leftrightarrow c_2}(r_2)$ is $CorrP_{c_2'}(r_2)$ except those nodes that are neither source nor target of a link in $CorrP_{c_2}(r_2)$ and those links whose source and target are not in $CorrP_{c_2}(r_2)$.*

The restricted pattern is necessary to handle cases like the following: Assume the axiom creates an additional element x (drawn with a dashed line in Fig. 5) in the class diagram, which also appears in $CorrP_{ca3}(r2)$ but not in $CorrP_{ca2}(r1)$. Then, an injective morphism from $CorrP_{ca3}(r2)$ to $CorrP_{ca2}(r1)$ would not exist, although the dependencies would still be the same. Therefore, all elements in $CorrP_{ca3}(r2)$ not directly connected to an element in $CorrP_{cb3}(r2)$ have to be ignored in order to detect a shared context.

The dependencies and shared contexts of a TGG rule are made explicit in *Rule Dependency Graphs* (RDG). For each TGG rule, one RDG is generated. It contains all correspondence nodes on the LHS of that rule, denoted as rounded rectangles in Fig. 6. All dependencies to these correspondence nodes are depicted using solid arrows from circles representing rules that produce matches for these correspondence nodes. Shared contexts are depicted using a dashed arrow from the correspondence node in the required rule (c_1 in Definition 3, denoted by a

[5] As a consequence of the fact that both r_1 and r_2 have a dependency to r_0, elements created by r_0 must match $CorrP_{c_1}(r_1)$ and $CorrP_{c_2}(r_2)$ at the same time. Therefore, the morphism between $CorrP_{c_1}(r_1)$ and $CorrPRes_{c_2' \leftrightarrow c_2}(r_2)$ matches nodes if they have the same type, one type is a subtype of the other, or both types have a common subtype.

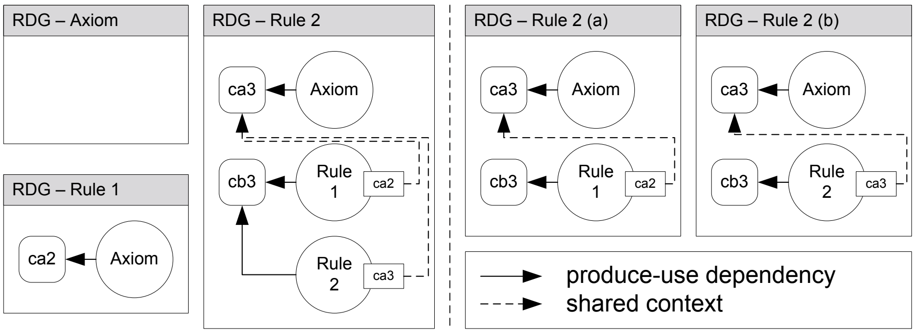

Fig. 6. Rule dependency graphs generated from the example TGG (Fig. 2)

rectangle) to the correspondence node in the current rule (c_2'). The RDGs of the example TGG are shown in the left half of Fig. 6.

Before generating *test case descriptions* from RDGs in the next step, the RDGs are simplified, so that only one dependency, i.e. one outgoing edge, is stored for each correspondence node. For example, *RDG - Rule 2* contains two dependencies for *cb3*. These are split to create *RDG - Rule 2 (a)* and *(b)*. If there are multiple correspondence nodes with more than one dependency, all combinations have to be built and the number of *simple* RDGs increases accordingly.

4.2 Deriving Test Cases from Rule Dependencies

A *Test Case Description* (TCD) is a sequence of TGG rules that also specifies the values of rule parameters and the bindings of LHS correspondence nodes to

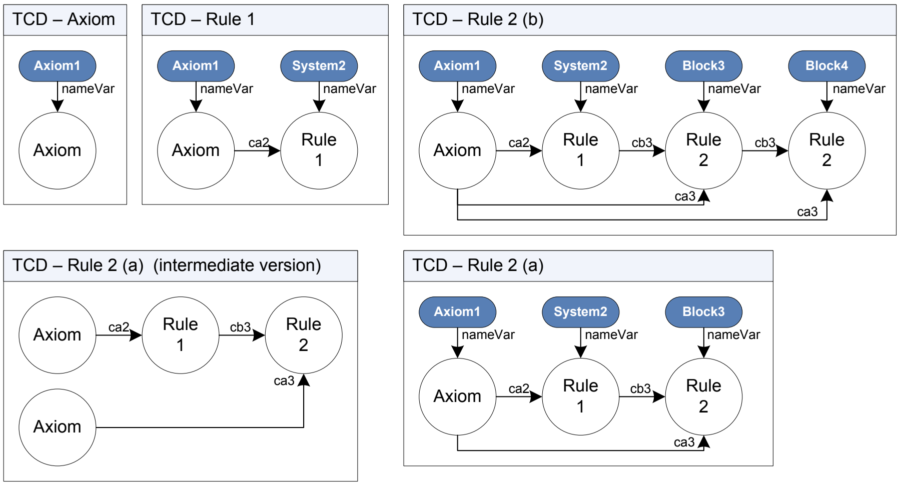

Fig. 7. Test case descriptions generated from the rule dependency graphs (Fig. 6)

previously created correspondence nodes. TCDs can be generated from *simple* RDGs in order to yield test cases with complete *rule dependency* coverage (cf. Sec. 4.3). Fig. 7 shows the TCDs generated from the *simple* RDGs in Fig. 6. The arrows denote the data flow of rule parameter values and created correspondence nodes. This generation algorithm works as follows:

```
 1  Set RDGs = {Set of simplified rule dependency graphs}
 2  Map TCDs = {} //maps TGG rules to their TCDs
 3
 4  while (RDGs is not empty) {
 5    rdg := remove an RDG from RDGs, where
 6      a TCD exists already for all required rules
 7
 8    tcd := create new TCD, add rule of rdg
 9
10    for each (TGG rule r required by rdg)  {
11      clone shortest TCD of r and insert into tcd
12    }
13    merge occurrences of rules according to RDG
14
15    sort rules by dependencies
16
17    add to TCDs
18  }
19
20  add values of primitive parameters to all TCDs
```

First, all simple RDGs are put into a set (line 1). Also, a map is created, which maps all TGG rules to their TCDs (line 2). Then, the TCDs are created in a loop. An RDG is selected, so that a TCD exists for all its required TGG rules (line 5).[6] A new TCD is created, which contains only the rule of the RDG. Then, rules have to be added to the TCD, which create the correspondence patterns required by the rule of the current RDG (lines 10-12). This is done by picking the shortest TCD from the map of already created TCDs and copying and inserting it into the current TCD. If no TCD has been created yet for a required rule, the TGG violates criterion 5 (cf. Sec. 2). After this step, the rules in the current TCD may not adhere to the shared contexts specified in the RDGs. For example, *TCD - Rule 2 (a) (intermediate version)* is the TCD generated from *RDG - Rule 2 (a)* after line 12. The axiom appears twice, once for rule 1 and once for rule 2, although both should use the result of a single axiom. Therefore, multiple occurrences of rules (or the axiom) have to be merged (line 13). After that, the rules have to be sorted by their dependencies, so that rules producing elements required by other rules come first (line 15). Finally, the new TCD is added to the map of TCDs. This process is repeated until all RDGs have been processed. After all TCDs have been generated, they have to be extended with values for rule parameters. Fig. 7 shows these final TCDs.

As a last step, the TCDs have to be executed. The rules in each TCD are applied successively. Rule parameters and LHS correspondence nodes are bound to the values specified in the TCDs. This produces sets of test cases to test a model transformation implementation. For example, executing *TCD - Rule 2(a)* produces the models shown in Fig. 3.

[6] In the first loop iteration, this is only the case for the axiom's RDG because it does not require any rules.

4.3 Completeness and Minimality of Test Cases

The test suite consisting of test cases specified by the TCDs generated from a TGG as described in Sec. 4.1 and Sec. 4.2 is complete w.r.t. rule dependency coverage. A test case *covers* a particular dependency $r_1 \to r_2^{c_2}$ if the rule sequence that built the test case contains at least r_1 and r_2, and the created elements of r_1 were bound to $CorrP_{c_2}(r_2)$ when r_2 was applied. Complete dependency coverage of generated test cases is ensured because the dependency analysis first finds all *produce-use* dependencies without considering shared contexts. If TCDs were then generated, all rule sequences with all possible rule dependencies would be generated. *TCD - Rule 2 (a) (intermediate version)* (cf. Fig. 7) would already be a final TCD. Then, all rules that occur multiple times would have to be combined in all possible ways to generate additional TCDs, *TCD - Rule 2 (a)* in the example. For more complex TGGs, a combinatorial explosion of the set of generated TCDs could be observed, e.g. if *TCD - Rule 2 (a) (intermediate version)* contains the axiom three times, this would result in four additional combinations. However, many combinations would not be applicable. By considering shared contexts, most non-applicable rule combinations are filtered out. If it is already known that a rule is only applicable in a certain context, all other contexts can be discarded.

A direct consequence of complete *rule dependency* coverage is complete *rule* coverage. If a test suite applies TGG rules so that all dependencies between rules are covered, then all rules of the TGG have to be applied, too, because every TGG rule depends on at least one other rule or the axiom (criterion 2).

Furthermore, each test case yielded from a generated TCD is *minimal* w.r.t. the rules required to test a particular dependency. *Minimal* means that if any rule except the last rule is removed from the TCD, the TCD would not be applicable anymore. This is ensured by the way in which TCDs are generated. Each TCD is generated for a rule and a particular context. Only those previously generated TCDs are added to this TCD, which contribute to the rule's context.

5 Evaluation

The presented algorithms for dependency analysis and test case generation have been implemented in QVT Operational and Java. They are available from the MDELab Update Site[7].

The new dependency-guided test case generation approach was verified by generating test cases for the same TGGs as in [10] and analyzing their specification coverage. The TGGs are: SDL2UML, which is slightly more complex than the example TGG (cf. Fig. 2); Automata2PLC, which is a transformation from automata models to a language for programmable logic controllers; and SystemDesk2AUTOSAR, which is a transformation from a tool-specific metamodel to AUTOSAR, a modeling standard from the automotive domain. Using the random test case generation approach (cf. Sec. 3), complete rule and rule

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

dependency coverage were achieved only for the SDL2UML and Automata2PLC TGGs. For SystemDesk2AUTOSAR, only 71% rule coverage and 19% rule dependency coverage were achieved using test models with more than 1000 model elements. With the new dependency-guided approach, complete rule and rule dependency coverage was achieved for all TGGs. Some of these test cases also uncovered previously unknown errors in the TGG.

Moreover, a weakness of the dependency analysis became visible. In the SystemDesk2AUTOSAR TGG, elements in the LHS of a rule have types that were too general. Several other rules produce elements that fit the correspondence pattern, which contains this general type. The dependency analysis detected these dependencies. However, due to the overall structure of the LHS of that rule, and in particular the connections between correspondence patterns, the rule was only applicable for particular subtypes. Not all dependencies detected by the dependency analysis yield applicable rule sequences. Still, one can argue that this is also an indication of a modeling error. Therefore, the rules were changed so that more concrete types are used and the problem disappeared. Another cause for non-executable test cases are OCL conditions, which are used to express structural application conditions or attribute constraints in a TGG rule. They may restrict applicability of a TGG rule but are not considered by the dependency analysis, yet. For these reasons, all non-applicable TCDs are also output by the testing framework. This assists the user in finding the reason why they are not executable.

Another advantage of the dependency-guided approach is that the generated test cases are minimal (cf Sec. 4.3). The largest test case for the SystemDesk2AUTOSAR TGG consists of only nine rule applications. Moreover, the rules are tested separately, i.e. there is a separate test case for each rule and each context. This helps in debugging the TGG implementation if errors are found. However, although the test cases themselves are usually small, the total number of test cases can be very large for complex TGGs. For example, 99 test cases were generated for the SystemDesk2AUTOSAR TGG. Yet, many test cases are already contained in others. In the example (cf. Fig. 7), *TCD - Rule 2(b)* contains all other TCDs. Therefore, it is possible to minimize the number of test cases by eliminating those test cases that are subsumed by others. This is done by a pairwise comparison of the generated test cases. The number of test cases for the SystemDesk2AUTOSAR TGG could be reduced from 99 to 68.

6 Related Work

A number of *conformance testing* approaches exists, which rely on graph transformation as a specification technique [2,8]. Instead of focusing on model transformation specifications and implementations, they are rather concerned with conformance testing of *behavioral specifications* w.r.t. (actual) behavior in refined models or (generated) code. There are some testing approaches proposed for *model transformation implementations*. Most *black-box* methods are concerned with generating qualified test input models (e.g. [15,4]) taking the input metamodel (and corresponding constraints) into consideration. For example, metamodel coverage is considered using data-partitioning techniques in [4]. It is required, for example, that models must contain representatives of association ends, which differ in their cardinalities. PaMoMo [1] is a high-level language for the specification of inter-model relationships, which can be used to check validity of models, derive model transformations, e.g. TGG rules, or derive transformation contracts for automated testing of transformation implementations. In contrast, *white-box* criteria are proposed in [11] to qualify test input models. The TGG conformance testing framework[10] generates conformance test cases using the model transformation specification as an "executable contract" generating not only test input models, but also expected results obtaining a complete oracle. In [3] the specification is used as partial oracle and no expected results are generated. Moreover, it proposes a new uniform framework, whereas the conformance testing framework [10] relies on TGGs as an existing model transformation specification technique for which several tools are already available.

Applicability criteria of graph transformation rule sequences are presented in [12]. If certain criteria are satisfied by the rules of a rule sequence, it can be decided statically whether the sequence is applicable or not. In general, though, the TGG rule sequences in the test case descriptions do not satisfy these criteria and, thus, their applicability cannot be checked statically using these results alone. Maybe specialized definitions can be formulated, which take the specific criteria of TGG rules (cf. Sec. 2) into account, but this is part of future work.

7 Conclusion

Model transformations play an important role in MDE. Triple graph grammars are an important representative of relational model transformations. The previously presented automatic conformance testing framework [10] can test conformance of a TGG implementation with its specification by automatically generating and executing test cases. Since this framework relies on a random generation approach, it cannot, in practice, achieve complete *specification coverage* for complex TGGs. The *dependency-guided* generation approach presented in this paper analyzes dependencies implicitly present in a TGG and generates test cases targeting these dependencies so that complete *specification coverage* is achievable in practice for well-formed TGGs. In addition, the generated test cases are minimal, which helps in debugging. The improved framework can now automatically generate high-quality test cases for conformance testing of TGG implementations.

In future work, the dependency analysis may be extended to analyze dependencies more thoroughly to cope with structural application conditions in TGG rules or even OCL constraints on the source and target metamodels.

References

1. de Lara, J., Guerra, E.: Inter-modelling with graphical constraints: Foundations and applications. EC-EASST 47 (2012)
2. Engels, G., Güldali, B., Lohmann, M.: Towards model-driven unit testing. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 182–192. Springer, Heidelberg (2007)
3. Fiorentini, C., Momigliano, A., Ornaghi, M., Poernomo, I.: A constructive approach to testing model transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 77–92. Springer, Heidelberg (2010)
4. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: Testing model transformations. In: Proc. of MoDeVa 2004, pp. 29–40. IEEE Computer Society Press (2004)
5. Giese, H., Hildebrandt, S., Lambers, L.: Toward bridging the gap between formal semantics and implementation of triple graph grammars. Technical Report 37, Hasso Plattner Institute at the University of Potsdam (2010)
6. Giese, H., Hildebrandt, S., Lambers, L.: Bridging the gap between formal semantics and implementation of triple graph grammars. Software and Systems Modeling, 1–27 (2012)
7. Golas, U., Lambers, L., Ehrig, H., Orejas, F.: Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. Theor. Comput. Sci. 424, 46–68 (2012)
8. Heckel, R., Mariani, L.: Automatic conformance testing of web services. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 34–48. Springer, Heidelberg (2005)
9. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In: Proc. of MDI 2012, pp. 22–31. ACM (2012)
10. Hildebrandt, S., Lambers, L., Giese, H., Petrick, D., Richter, I.: Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 238–253. Springer, Heidelberg (2012)
11. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations – first experiences using a white box approach. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
12. Lambers, L., Ehrig, H., Taentzer, G.: Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences. In: de Lara, J., Ermel, C., Heckel, R. (eds.) Proc. GT-VMT 2008, vol. 10, EC-EASST, Budapest (2008)
13. Lambers, L., Hildebrandt, S., Giese, H., Orejas, F.: Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case. In: Proceedings of BX 2012, vol. 49, pp. 1–16. EC-EASST (2012)
14. Schürr, A., Klar, F.: 15 years of triple graph grammars: research challenges, new contributions, open problems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
15. Sen, S., Baudry, B., Mottu, J.-M.: Automatic model generation strategies for model transformation testing. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 148–164. Springer, Heidelberg (2009)