# Efficient Model Synchronization by Automatically Constructed Repair Processes
## Extended Version

Lars Fritsche[1], Jens Kosiol[2], Andy Schürr[1], and Gabriele Taentzer[2]

[1] TU Darmstadt
{lars.fritsche,andy.schuerr}@es.tu-darmstadt.de
[2] Philipps-Universität Marburg
{kosiolje,taentzer}@mathematik.uni-marburg.de

**Abstract.** Model synchronization, i.e., the task of restoring consistency between two interrelated models after a model change, is a challenging task. Triple Graph Grammars (TGGs) specify model consistency by means of rules. They can be used to automatically derive specifications of edit operations for single models and repair rules that propagate model changes to related models. To support model (re-)synchronization activities more effectively, a construction mechanism for *short-cut* rules has been recently developed. They describe consistency-preserving complex edit operations across model boundaries. We show that edit and repair rules can be derived from *short-cut* rules. As proof of concept, we implemented the construction and application of *short-cut* edit and repair rules in eMoflon. Our evaluation shows that *short-cut*-rule-based repair processes have considerably decreased data loss and improved runtime compared to former model synchronization processes in eMoflon.

**Keywords:** Model Synchronization · Triple Graph Grammars · Short-Cut Rule

## 1 Introduction

Model-driven engineering has become an important technique to cope with the increasing complexity of modern software systems. In the field of Concurrent Engineering [7], for example, products are no longer realized in series but allow parallel tasks. Each of these tasks has its view onto the product and, as a view evolves, it may become inconsistent with the other ones. Keeping views synchronized by checking and preserving their consistency can be a challenging problem which is not only subject to ongoing research but also of practical interest for industrial applications such as stated above.

Triple Graph Grammars (TGGs) [23] are a declarative, rule-based bidirectional transformation approach that aims to synchronize models stemming from different views (usually called *domains* in the TGG literature). Their purpose is to define a consistency relationship between pairs of models in a rule-based manner by defining traces between their elements. Given a finite set of rules that

define how both models co-evolve, a TGG can be automatically *operationalized* into *source* and *forward rules*. The source rules of an operationalized TGG can be used to build up models of one domain while forward rules translate them to models of the other domain, thereby establishing traces between their elements. From a synchronization point of view, source rules specify edit operations to change one model while forward rules specify repair operations to synchronize model changes with one another [23,18,15]. Even though both, the translation and the synchronization process, are formally defined and sound, there are in fact several practical issues that arise for model synchronization from (potentially transitive) dependencies between rule applications: To synchronize changed models, popular TGG approaches do not always fix inconsistencies locally but revert all dependent rule applications and start a retranslation process. However, this kind of synchronization often deletes and recreates a lot of model elements to reestablish model consistency, potentially losing information that is local to just one model and wasting processing time. Existing solutions for this problem are rather ad hoc and come without any guarantee to reestablish the consistency of modified models [11,13].

As a new solution to this synchronization problem, we derive *repair rules* from *short-cut* rules [8] that we recently introduced to handle complex consistency-preserving model updates more effectively and efficiently. The construction of *short-cut* rules is a kind of sequential rule composition that allows to replace a rule application with another one while preserving involved model elements (instead of deleting and re-creating them). We used *short-cut* rules to describe model changes exchanging one edit step by another one. Since in this paper we want to use *short-cut* rules for model synchronization as well, they have to be operationalized into *source* and *forward* rules.

Our formal contributions (in Sect. 4) are two-fold: As *short-cut* rules may be non-monotonic, i.e., may be deleting, we formalize the operationalization of non-monotonic TGG rules which decomposes short-cut rules into (semantically equivalent sequences of) source (edit) and forward (repair) rules. Moreover, we obtain sufficient conditions under which an application of a *short-cut* rule preserves the consistency of related pairs of models. This was left to future work in [8]. Together, this constitutes the correctness of our approach using operationalized *short-cut* rules for model synchronization.

Practically, we implement our synchronization approach in eMoflon [20], a state-of-the-art bidirectional graph transformation tool, and evaluate it (Sect. 5). The results show that the construction of *short-cut* repair rules enables us to react to model changes in a less invasive way by preserving information and increasing the performance. We thus contribute to a more comprehensive research trend in the bx-community towards *Least Change* synchronization [5]. Before presenting these results in detail, we illustrate our approach using an example in Sect. 2 and recall some preliminaries in Sect. 3. Finally, we discuss related work in Sect. 6 and conclude with pointers to future work in Sect. 7. In an appendix, we present additional preliminaries (Appendix A), all proofs (Appendix B), and the rule

set used for our evaluation, including more complex examples (Appendices C and D).

## 2 Introductory Example

We motivate the use of *short-cut* repair processes by synchronizing a Java AST (abstract syntax tree) model and a custom documentation model. For model synchronization, we consider a Java AST model as *source* model and its documentation model as *target* model, i.e., changes in a Java AST model have to be transferred to its documentation model. There are correspondence links in between such that both models become correlated.

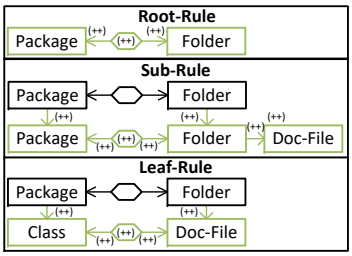



**Fig. 1.** Example: TGG Rules

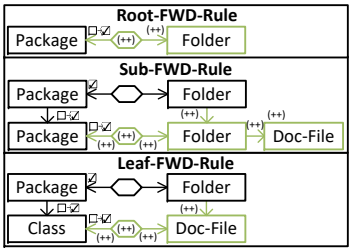



**Fig. 2.** Example: TGG Forward Rules

*TGG rules.* Figure 1 shows the rule set of our running example consisting of three TGG rules: *Root-Rule* creates a root *Package* together with a root *Folder* and a correspondence link in between. This rule has an empty precondition and only creates elements which are depicted in green and with the annotation (++). *Sub-Rule* creates a *Package* and *Folder* hierarchy given that an already correlated *Package* and *Folder* pair exists. Finally, *Leaf-Rule* creates a *Class* and a *Doc-File* under the same precondition as *Sub-Rule*.

These rules can be used to generate consistent triple graphs in a synchronized way consisting of source, correspondence, and target graph. A more general scenario of model synchronization is, however, to restore the consistency of a triple graph that has been altered on just one side. For this purpose, each TGG rule has to be operationalized to two kinds of rules: *source* rules enable changes of source models which is followed by translating this model to the target domain with *forward* rules. As *source* rules for single models are just projections of TGG rules to one domain, we do not show them explicitly.

*Forward translation rules.* Figure 2 depicts the *forward* rules. Using these rules, we can translate the Java AST model depicted on the source side of the triple graph in Fig. 3 (a) to a documentation model such that the result is the complete graph in Fig. 3 (a). To obtain this result we apply *Root-FWD-Rule* at the root *Package*, *Sub-FWD-Rule* at *Packages* `p` and `subP`, and finally *Leaf-FWD-Rule* at *Class* `c`. To guide the translation process, context elements that have already

been translated are annotated with ☑ in *forward* rules. A formerly created source element gets the marking □ → ☑ to indicate that applying the rule will mark this element as translated; a formalization of this marking is given in [19]. Note that *Root-FWD-Rule* can always be applied when *Sub-FWD-Rule* is applicable which can lead to untranslated edges. For simplicity, we assume that the correct rule is applied which in praxis can be achieved through negative application conditions [14].

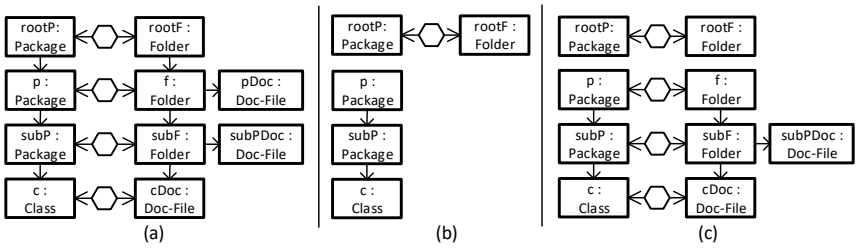



**Fig. 3.** Exemplary Synchronization Scenario

*Model synchronization.* Given the triple graph in Fig. 3 (a), a user might want to change a sub *Package* such as `p` to be a root *Package*, e.g., as could be the case when the project is split up into multiple projects. Since `p` was created and translated as a sub *Package* rather than a root element, this change introduces an inconsistency. To resolve this issue, one approach is to revert the translation of `p` into `f` and re-translate `p` with an appropriate translation rule such as *Root-FWD-Rule*. Reverting the former translation step may lead to further inconsistencies as we remove elements that were needed as context elements by other rule applications. The result is a reversion of all translation steps except for the first one which translated the original root element. The result is shown in Fig. 3 (b). Now, we can re-translate the unmarked elements yielding the result graph in (c). This example shows that this synchronization approach may delete and re-create a lot of similar structures which appears to be inefficient. Second, it may lose information that exists on the target side only, e.g., a use case may be assigned to a document which does not have a representation in the corresponding Java project.

*Model synchronization with short-cut repair.* In [8] we introduced short-cut rules as a kind of rule composition mechanism that allows to replace a rule application by another one while preserving elements (instead of deleting and re-creating them). In our example, *Root-Rule* and *Sub-Rule* overlap in elements as the first rule can be completely embedded into the latter one. Figure 4 depicts two possible short-cut rules based on *Root-Rule* and *Sub-Rule*. While the upper short-cut rule replaces *Root-Rule* with *Sub-Rule*, the lower short-cut rule replaces *Sub-Rule* with *Root-Rule*. Both short-cut rules preserve the model elements on both sides and solely create elements that do not yet exist (++), or delete those depicted

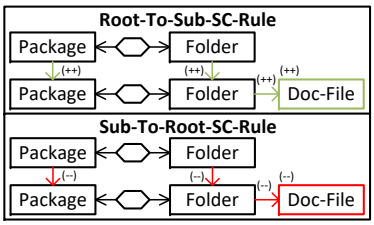



**Fig. 4.** Short-cut rules

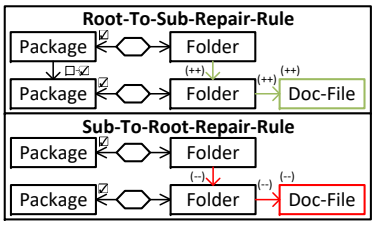



**Fig. 5.** Repair rules

in red and annotated with (--). They are constructed by overlapping both original rules such that each created element that can be mapped to the other rule becomes context and as such, is not touched. When a created element cannot be mapped because it only appears in the replacing rule, it is created. Consequently, an element is deleted if the created element only appears in the replaced rule. Finally, context elements occurring in both rules appear also in the short-cut rule while overlapped context elements appear only once. Using *Sub-To-Root-SC-Rule* enables the user to transform the triple graph in Fig. 3 (a) directly to the one in (c).

Yet, these rules can still not cope with the change of a single model since short-cut rules transform both models at once as TGG rules usually do. Hence, in order to be able to handle the deleted edge between `rootP` and `p`, we have to forward operationalize short-cut rules, thereby obtaining *short-cut repair* rules. Figure 5 depicts the resulting *short-cut repair* rules derived from *short-cut* rules in Fig. 4. A non-monotonic TGG-rule is forward operationalized by removing deleted elements from the rule's source graphs as they should not be present after a source rule application. *Short-cut repair* rules allow to propagate source graph changes directly to target graphs to restore consistency. In our example, after having transformed Package `p` into a root element, the rule of choice is *Sub-To-Root-Repair-Rule* which transforms Folder `f` in Fig. 3 (a) into a root element and deletes the superfluous *Doc-File*. The result is again the consistent triple graph depicted in Fig. 3 (c). This repair allows to skip the costly reversion process with the intermediate result in Fig. 3 (b). Note that applying *Sub-To-Root-Repair-Rule* at arbitrary matches may have undesired consequences: One could, e.g., delete the edge between two *Folders* even if the matched *Packages* are still connected. Our Theorem 8 characterizes matches where such violations of the language of the grammar cannot happen. In our implementation, we exploit an incremental pattern matcher to identify valid matches. Using suitable *negative application conditions* [6] would be an alternative approach.

## 3 Preliminaries

To understand our formal contributions, we assume familiarity with the basics of double-pushout rewriting in graph transformation and, more generally in adhesive categories [6,17] as well as the definition of TGGs and in particular, their

operationalizations [23]. Here, we recall non-basic preliminaries for our work which are the construction of short-cut rules, the notion of sequential independence, and a (simple) categorical definition of partial maps.

In [8], we introduced short-cut rules as a new way of sequential composition for monotonic rules. Given an inverse rule of a monotonic rule (i.e., a rule that only deletes) and a monotonic rule, a short-cut rule combines their respective actions into a single rule. Its construction allows to identify elements that are deleted by the first rule as re-created by the second one. These elements are preserved in the resulting short-cut rule. A *common kernel*, i.e., a common subrule of both, serves to identify how the two rules overlap and which elements are preserved instead of being deleted and re-created. We recall their construction since our construction of repair rules is based on it. Examples are depicted in Fig. 4.

**Definition 1 (Short-cut rule).** *In an adhesive category* $\boldsymbol{C}$*, given two monotonic rules* $r_i : L_i \hookrightarrow R_i$*,* $i = 1, 2$*, and a common kernel rule* $k : L_\cap \hookrightarrow R_\cap$ *for them, the* short-cut rule $r_1^{-1} \ltimes_k r_2 := (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ *is computed by executing the following steps depicted in Figs. 6 and 7:*

1. *The union* $L_\cup$ *of* $L_1$ *and* $L_2$ *along* $L_\cap$ *is computed as pushout* (2)*.*
2. *The LHS* $L$ *of the short-cut rule* $r_1^{-1} \ltimes_k r_2$ *is computed as pushout* (3*a*)*.*
3. *The RHS* $R$ *of the short-cut rule* $r_1^{-1} \ltimes_k r_2$ *is computed as pushout* (3*b*)*.*
4. *The interface* $K$ *of the short-cut rule* $r_1^{-1} \ltimes_k r_2$ *is computed as pushout* (4)*.*
5. *Morphisms* $l : K \to L$ *and* $r : K \to R$ *are obtained by the universal property of* $K$*.*

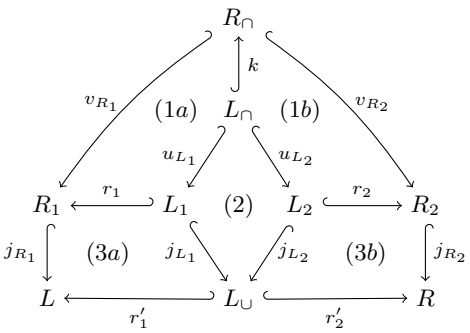



**Fig. 6.** Construction of LHS and RHS of short-cut rule $r_1^{-1} \ltimes_k r_2$

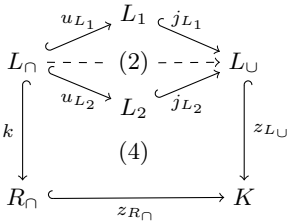
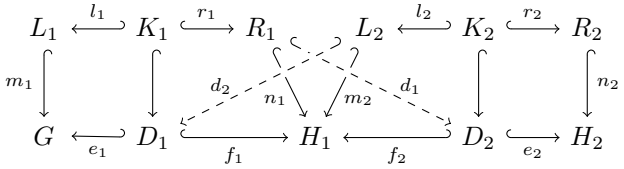



**Fig. 7.** Construction of interface $K$ of $r_1^{-1} \ltimes_k r_2$

Sequential independence of two rule applications intuitively means that none of these applications enables the other one. This implies that the order of their application may be switched. The definition of sequential independence can be extended to a sequence of rule applications longer than 2. In Theorem 8, we will use this to identify language-preserving applications of short-cut rules.

**Definition 2 (Sequential independence).** *Given two rules* $p_i = (L_i \overset{l_i}{\hookleftarrow} K_i \overset{r_i}{\hookrightarrow} R_i)$ *with* $i = 1, 2$, *two direct transformations* $G \Rightarrow_{p_1,m_1} H_1$ *and* $H_1 \Rightarrow_{p_2,m_2} H_2$ *via the rules* $r_1$ *and* $r_2$ *are* sequentially independent *if there exist two morphisms* $d_1 : R_1 \to D_2$ *and* $d_2 : L_2 \to D_1$ *as depicted below such that* $n_1 = f_2 \circ d_1$ *and* $m_2 = f_1 \circ d_2$.




*Given rules* $p = (L \hookleftarrow K \hookrightarrow R)$ *and* $p_i = (L_i \hookleftarrow K_i \hookrightarrow R_i)$ *with* $1 \leq i \leq t$, *a transformation* $G_t \Rightarrow_{p,m} H$ *is* sequentially independent from a sequence of transformations $G_0 \Rightarrow_{p_1,m_1} G_1 \Rightarrow_{p_2,m_2} \cdots \Rightarrow_{p_t,m_t} G_t, t \geq 2$ *if first,* $G_t \Rightarrow_{p,m} H$ *and* $G_{t-1} \Rightarrow_{p_t,m_t} G_t$ *are sequentially independent and then, the arising transformations* $G_{t-1} \Rightarrow_{p,e_t \circ d_2^t} G'_t$ *and* $G_{t-2} \Rightarrow_{p_{t-1},m_{t-1}} G_{t-1}$ *are sequentially independent and so forth back to the transformations* $G_0 \Rightarrow_{p_1,m_1} G_1$ *and* $G_1 \Rightarrow_{p,e_2 \circ d_2^2} G'_2$ *(where* $e_i : D_i \hookrightarrow G_{i-1}$ *is given by the transformation and* $d_2^i : L \hookrightarrow D_i$ *exists by sequential independence as in the figure above).*

To formalize the application of non-monotonic TGG rules, we need to consider triple graphs with partial morphisms from correspondence to source (or target) graphs. For expressing such triple graphs categorically, we recall a simple definition of partial morphisms [22] to be used in Sect. 4.1. An elaborated theory of triple graphs with partial morphisms is out of scope of this paper.

**Definition 3 (Partial morphism. Commuting square with partial morphisms).** *A* partial morphism *a from an object A to an object B is a(n equivalence class of) span(s)* $A \overset{\iota_A}{\hookleftarrow} A' \overset{a}{\to} B$ *where* $\iota_A$ *is a monomorphism (denoted by* $\hookrightarrow$*). A partial morphism is denoted as* $a : A \dashrightarrow B$; $A'$ *is called the* domain *of a. A diagram with two partial morphisms a and c as depicted as square* (1) *in Fig. 8 is said to be* commuting *if there exists a (necessarily unique) morphism* $x : A' \to C'$ *such that both arising squares* (2) *and* (3) *in Fig. 9 commute.*

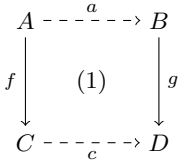



**Fig. 8.** Square of partial morphisms

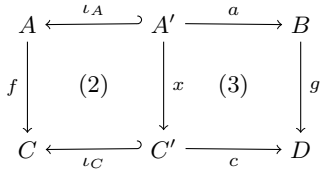



**Fig. 9.** Commuting square of partial morphisms

## 4 Constructing Language-Preserving Repair Rules

The general idea of this paper is to use *short-cut repair* rules allowing an optimized model synchronization process based on TGGs. To this end, we operationalize short-cut rules being constructed from the rules of a given TGG. Since those rules are not necessarily monotonic, we generalize the well-known operationalization of TGG rules to the non-monotonic case and show that the basic property is still valid: An application of a source rule followed by an application of the corresponding forward rule is equivalent to applying the original rule instead. This is the content of Sect. 4.1. Constructing *short-cut* rules in [8], we identified the following problem: Applying a short-cut rule derived from rules of a given grammar might lead to an instance that is not part of the language defined by that grammar. Therefore, in Sect. 4.2, we provide sufficient conditions for applications of short-cut rules leading to instances of the grammar-defined language only. Combining both results ensures the correctness of our approach, i.e., a *short-cut* repair rule actually propagates a model change from the source to the target model if it is correctly matched.

### 4.1 Operationalization of Generalized TGG Rules

Since the operationalization of TGG rules has been introduced for monotonic rules only, we extend the theory to general triple rules and, moreover, allow for partial morphisms from correspondence to source and target graph in triple graphs. We split a rule on triple graphs into a *source rule* that only affects the source part and a *forward rule* that affects correspondence and target part.

**Definition 4 (TGG rule).** *Let the category of triple graphs and graph morphisms be given. A triple rule $p$ is a span of triple graph morphisms*

$$p{=}((L_S \xleftarrow{\sigma_L} L_C \xrightarrow{\tau_L} L_T) \xleftarrow{(l_S,l_C,l_T)} (K_S \xleftarrow{\sigma_K} K_C \xrightarrow{\tau_K} K_T) \xhookrightarrow{(r_S,r_C,r_T)} (R_S \xleftarrow{\sigma_R} R_C \xrightarrow{\tau_R} R_T))$$

*which, wherever possible, are abbreviated by*

$$p{=}(L_{SCT} \xleftarrow{(l_S,l_C,l_T)} K_{SCT} \xhookrightarrow{(r_S,r_C,r_T)} R_{SCT}) \quad .$$

*Rules $p_S$ and $p_F$ are called* source rule *and* forward rule *of $p$.*

$$p_S{=}((L_S \leftarrow \emptyset \rightarrow \emptyset) \xleftarrow{(l_S,id_\emptyset,id_\emptyset)} (K_S \leftarrow \emptyset \rightarrow \emptyset) \xhookrightarrow{(r_S,id_\emptyset,id_\emptyset)} (R_S \leftarrow \emptyset \rightarrow \emptyset)) \quad ,$$

$$p_F{=}(R_S L_{CT} \xleftarrow{(id_{R_S},l_C,l_T)} R_S K_{CT} \xrightarrow{(id_{R_S},r_C,r_T)} R_{SCT})$$

*with $\emptyset$ being the empty graph. In $R_S L_{CT} = (R_S \leftarrow\!\!- L_C \xhookrightarrow{\tau_L} L_T)$, the morphism from $L_C$ to $R_S$ may be partial and is defined by the span $(L_C \xhookleftarrow{l_C} K_C \xhookrightarrow{r_S \circ \sigma_K} R_S)$ with $\sigma_K : K_C \hookrightarrow R_C$.* Target *and* backward rules *$p_T$ and $p_B$ are defined symmetrically in the other direction.*

*Given a TGG, a* short-cut repair rule *is a forward rule $p_F$ of a short-cut rule $p = r_1^{-1} \ltimes_k r_2$ where $r_1, r_2$ are (monotonic) rules of the TGG, i.e., a repair rule is an operationalized short-cut rule.*

The above definition is motivated by our application scenario, i.e., the case where a user edits the source (or target) model independently of the other parts. The partial morphism in the forward rule reflects that a model change may introduce a situation where the result is no longer a triple graph. A deleted source element may have a preimage in the correspondence graph that is not deleted as well. In the example *short-cut* rules in Fig. 4, this problem does not occur since edges are deleted only. But in general, this definition of $p_S$ has the disadvantage that often, $p_S$ is not applicable to any triple graph since the result would not be one.

In practical applications, however, the source rule specifies a user edit action that is performed on the source part only, ignoring correspondence and target graphs. The fact that the result is not a triple graph any longer is not a technical problem. A missing source element that should be referenced by a correspondence element gives information about a location that needs some repair. Therefore, we define the application of a source rule such that the resulting triple graph is allowed to be partial. Furthermore, forward rules may be applied to partial triple graphs allowing for dangling correspondence relations.

**Definition 5 (Constructing an operationalized rule application).** *Let a triple graph rule* $p = (L_{SCT} \xleftarrow{(l_S,l_C,l_T)} K_{SCT} \xrightarrow{(r_S,r_C,r_T)} R_{SCT})$ *with source rule* $p_S$ *and forward rule* $p_F$ *be given. An operationalized rule application* $G \Rightarrow_{p_S,m_S} G' \Rightarrow_{p_F,m_F} H$ *is constructed as follows:*

1. *The rule* $p_S^{\mathrm{pr}} = L_S \xleftarrow{l_S} K_S \xrightarrow{r_S} R_S$ *is the* projection *of* $p_S$ *to its source part.*
2. *Given a match* $m_S^{\mathrm{pr}}$ *for* $p_S^{\mathrm{pr}}$*, construct the transformation* $t_S^{\mathrm{pr}} : G_S \Rightarrow_{p_S^{\mathrm{pr}},m_S^{\mathrm{pr}}} H_S$*, called* source application *and inducing the span* $G_S \xleftarrow{f_S} D_S \xhookrightarrow{g_S} H_S$*.*
3. *The transformation* $t_S^{\mathrm{pr}}$ *can be extended to the transformation* $t_S : G = (G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T) \Rightarrow_{p_S,m_S} G' = (H_S \dashleftarrow G_C \xrightarrow{\tau_G} G_T)$ *via* $p_S$ *at match* $m_S$*. The partial morphism* $G_C \dashrightarrow H_S$ *is given as the span* $G_C \hookleftarrow G'_C \to H_S$ *that arises as pullback of the co-span* $G_C \to G_S \hookleftarrow D_S$ *as depicted in Fig. 10, i.e., as morphism* $g_S \circ p_D : G_C \dashrightarrow H_S$ *with domain* $G'_C$*.*
4. *Given co-match* $n_S : R_S \hookrightarrow H_S$ *and matches* $m_X : L_X \hookrightarrow G_X$ *with* $X \in \{C,T\}$ *such that both arising squares are commuting, i.e.,* $m_F = (n_S, m_C, m_T)$ *is a morphism of partial triple graphs, construct transformation* $t_F : G' \Rightarrow_{p_F,m_F} H = (H_S \xleftarrow{\sigma_H} H_C \xrightarrow{\tau_H} H_T)$*, called* forward application*, using transformations* $G_X \Rightarrow_{p_X,m_X} H_X$ *for* $X \in \{C,T\}$ *if they exist and if there are morphisms* $\sigma'_D : D_C \to H_S$ *and* $\tau_D : D_C \to D_T$ *such that* $H_S D_C D_T \hookrightarrow H_S G_C G_T$ *and* $R_S K_C K_T \hookrightarrow H_S D_C D_T$ *are triple morphisms.*

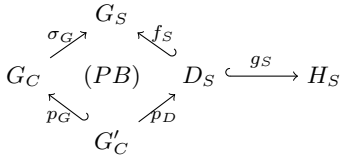



**Fig. 10.** Retrieval of partial morphism $G_C \dashrightarrow H_S$

In the setting of this paper, it is enough to allow for partial morphisms only in the input graph and not in the output graph of a forward rule application. Intuitively this means that such an application deletes those elements from the correspondence graph that could not be mapped to elements in the source graph any longer and additionally deletes the preimages in the correspondence graph of all deleted elements from the target graph as well (if there are any). The next lemma states that the application of a source rule is well-defined, i.e., that the mentioned partial morphism actually exists.

**Lemma 6 (Correctness of application of source rules).** *Let a (non-monotonic) triple graph rule*

$$p{=}(L_{SCT} \xleftarrow{(l_S,l_C,l_T)} K_{SCT} \xrightarrow{(r_S,r_C,r_T)} R_{SCT})$$

*with source rule $p_S$ and projection $p_S^{\mathrm{pr}}$ to the source part be given. Given a match $m_S$ for $p_S$ to a triple graph $G = (G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T)$ such that $G_S \Rightarrow_{p_S^{\mathrm{pr}},m_S} H_S$, the partial morphism $D_C \dashrightarrow H_S$ as described in Definition 5 exists.*

The next theorem states that a sequential application of a source and a forward rule indeed coincides with an application of the original rule as long as the matches are consistent. This means that the forward rule has to match the RHS $R_S$ of the source rule again and the LHS $L_C$ of the correspondence rule needs to be matched in such a way that all elements not belonging to the domain of the partial morphism from correspondence to source part in the input model are deleted. The forward rule application defined in Definition 5 fulfills this condition by construction.

**Theorem 7 (Synthesis of rule applications).** *Let a triple graph rule $p$ with source and forward rules $p_S$ and $p_F$ be given. If there are applications $G \Rightarrow_{p_S,m_S} G'$ with co-match $n_S$ and $G' \Rightarrow_{p_F,m_F} H$ with $m_F = (n_S, m_C, m_T)$ as constructed above, then there is an application $G \Rightarrow_{p,m} H$ with $m = (m_S, m_C, m_T)$.*

### 4.2 Language-Preserving Short-Cut Rules

In this section we identify sufficient conditions for an application of a short-cut rule that guarantee the result to be an element of the language of the original grammar. Since our conditions apply to arbitrary adhesive categories and are not specific for TGGs, we present the result in its general form.

**Theorem 8 (Characterization of valid applications).** *In an adhesive category $\mathcal{C}$, given a sequence of transformations*

$$G \Rightarrow_{r,m} G_0 \Rightarrow_{p_1,m_1} G_1 \Rightarrow_{p_2,m_2} \cdots \Rightarrow_{p_t,m_t} G_t \Rightarrow_{r^{-1} \ltimes_k r',m_{sc}} H$$

*with rules $p_1, \ldots, p_t$ and $r^{-1} \ltimes_k r'$ being the short-cut rule of monotonic rules $r : L \hookrightarrow R$ and $r' : L' \hookrightarrow R'$ along a common kernel $k$, there is a match $m'$ for $r'$ in $G$ and a transformation sequence*

$$G \Rightarrow_{r',m'} G'_1 \Rightarrow_{p_1,m'_1} \ldots G'_{t-1} \Rightarrow_{p_t,m'_t} H \ ,$$

*provided that*

1. *the application of* $r^{-1} \ltimes_k r'$ *with match* $m_{sc}$ *is sequentially independent of the sequence of transformations* $G_0 \Rightarrow_{p_1,m_1} G_1 \Rightarrow_{p_2,m_2} \cdots \Rightarrow_{p_t,m_t} G_t$ *and*
2. *the thereby implied match* $m'_{sc}$ *for* $r^{-1} \ltimes_k r'$ *in* $G_0$*, restricted to the RHS* $R$ *of* $r$*, equals the co-match* $n : R \hookrightarrow G_0$ *of the transformation* $G \Rightarrow_{r,m} G_0$ *(i.e.,* $m'_{sc} \circ j_R = n$ *where* $j_R$ *embeds* $R$ *into the LHS of* $r^{-1} \ltimes_k r'$ *as in Fig. 6).*

*In particular, given a grammar* $GG = (\mathcal{R}, S)$ *such that* $r, r', p_1, \ldots, p_t \in \mathcal{R}$ *and* $G \in \mathcal{L}(GG)$*, then* $H \in \mathcal{L}(GG)$*.*

Independence of the short-cut rule application $t_{sc} : G_t \Rightarrow_{r^{-1} \ltimes_k r', m_{sc}} H$ from the preceding transformation sequence $t : G \Rightarrow G_t$ requires the existence of morphisms in two directions: morphisms $d_2^i$ from the LHS of the short-cut rule to the context objects $D_i$ arising in $t$ and morphisms $d_1^i$ from the right-hand sides $R_i$ of the rules $p_i$ to the context object of $t_{sc}$ (shifted further and further to the beginning of the sequence). In the case of (typed triple) graphs, the existence of morphisms $d_2^i$ ensures that none of the rule applications in $t$ enabled the transformation $t_{sc}$. The existence of morphisms $d_1^i$ ensures that the transformation $t_{sc}$ does not delete structure needed to perform the transformation sequence $t$.

*Application to model synchronization.* The results in Theorems 7 and 8 are the formal basis for an automatic construction of repair rules. Theorem 7 ensures that a suitable edit action followed by application of a repair rule at the right match is equivalent to the application of a short-cut rule. Thus, whenever an edit action on the source model (or symmetrically the target model) corresponds to the source-action (target-action) of a short-cut rule, application of the corresponding forward (backward) rule synchronizes the model again. Since the language of a TGG is defined by its rules, every valid model can be reached from every other valid model by inverse application of some of the rules of the grammar followed by normal application of some rules. Often, edit actions are rather small steps (or at least consist of those). Thus, it is not unreasonable to expect that many typical edit actions can be realized as short-cut rules as these formalize the inverse application of a rule followed by application of a normal one. Theorem 8 characterizes the matches for short-cut rules at which application stays in the language of the TGG. For operational short-cut rules, this can either be used for detecting invalid edit actions or determining valid matches for synchronizing forward rules.

## 5 Implementation and Evaluation

*Implementation.* Our implementation[3] of an optimized model synchronizer is based on the existing EMF-based general purpose graph and model transformation tool eMoflon [20]. It offers support for rule-based unidirectional and bidirectional graph transformations where the latter is based on TGGs. To support an effective model synchronizer, we automatically calculate a small but

[3] Both the implementation and evaluation workspace can be accessed via https://github.com/Arikae00/FASE19_eMoflon-evaluation.

useful subset of all possible short-cut rules. This is done by overlapping as many created elements as possible and only varying in the way that context elements are mapped onto each other. These selected short-cut rules are operationalized to get repair rules that allow us to repair broken links similar to our example in Sect. 2. The model synchronization process is based on an *incremental graph pattern matcher* that tracks all matches that dis-/appear due to model changes. Thus, it offers the ability to react to model changes without the need to recompute matches from scratch. Our implementation uses this technique by processing all those matches marked as broken by the pattern matcher after a model change. A broken match is the starting point to find a repair match as it is defined by the co-match of the performed model change and has to be extended. If the pattern matcher can extend a broken match to a repair match, the corresponding *short-cut* repair rule can be applied. Otherwise, we fall back to the old synchronization strategy of revoking the current step. This completely automatized synchronization process ensures that we are able to restore consistency as long as the edited domain model still resides in the language of our TGG.

*Evaluation.* Our experimental setup consists of 23 TGG rules (shown in Appendix C) that specify consistency between Java AST and custom documentation models and 37 short-cut rules derived from our TGG rule set. A small modified excerpt of this rule set was given in Sect. 2. For this evaluation, however, we define consistency not only between *Package* and *Folder* hierarchies but also between type definitions, e.g., *Classes* and *Interfaces*, and *Methods* with their corresponding documentation entries. We extracted five models from Java projects hosted on Github using the tool MoDisco [4] and translated them into our own documentation structure. Also, we generated five synthetic models consisting of n-level *Package* hierarchies with each non-leaf *Package* containing five sub-*Packages* and each leaf *Package* containing five *Classes*. Given such Java models, we refactored each model in three different scenarios such as by moving a *Class* from one *Package* to another or completely relocating a *Package*. Then we used eMoflon to synchronize these changes in order to restore consistency to the documentation model, with and without *repair rules*.

These synchronization steps are subject to our evaluation and we pose the following research questions: **(RQ1)** *For different kinds of changes, how many elements can be preserved that would otherwise be deleted and recreated?* **(RQ2)** *How does our new approach affect the runtime performance?* **(RQ3)** *Are there specific scenarios in which our approach performs especially good or bad?*

*Repair rules* were developed to avoid unnecessary deletions of elements by reverting too many rule applications in order to restore consistency as shown exemplary in Sect. 2. This means that model changes where our approach should perform especially good, have to target rule applications close to the beginning of a rule sequence as this possibly renders many rule applications invalid. This means that altering a root *Package* by creating a new *Package* as root would imply that many rule applications have to be reverted to synchronize the changes correctly (Scenario 1). In contrast, our approach might perform poorly when a model change does not inflict a large cascade of invalid rule applications. Hence,

we move *Classes* between *Packages* to measure if the effort of applying *repair rules* does infer a performance loss when both the new and old algorithm do not have to repair many broken rule applications (Scenario 2). Finally, we simulate a scenario between the first two by relocating leaf *Packages* (Scenario 3).

**Table 1.** Legacy vs. new synchronizer – Time in sec. and number of created elements

| | Both | | Legacy Synchronization | | | | | | Synchro. by Repair Rules | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Trans. | | Scen. 1 | | Scen. 2 | | Scen. 3 | | Scen. 1 | | Scen. 2 | | Scen. 3 | |
| Models | Sec | Elts | Sec | Elts | Sec | Elts | Sec | Elts | Sec | Elts | Sec | Elts | Sec | Elts |
| lang.List | 0.3 | 25 | 0.2 | 20 | – | – | 0.06 | 5 | 0.2 | 0 | – | – | 0.03 | 0 |
| tgg.core | 6.4 | 1.6k | 39 | 1.6k | 3.8 | 99 | 0.64 | 17 | 0.8 | 0 | 0.11 | 0 | 0.05 | 0 |
| modisco.java | 9.9 | 3.2k | 228 | 3.3k | 18.6 | 192 | 3.6 | 33 | 2.5 | 0 | 0.2 | 0 | 0.09 | 0 |
| eclipse.graphiti | 20.7 | 6.5k | 704 | 6.5k | 63.9 | 490 | 5.65 | 25 | 6.1 | 0 | 0.21 | 0 | 0.09 | 0 |
| eclipse.compare | 10.74 | 3.8k | 83 | 3.7k | 3.1 | 76 | 2.36 | 47 | 0.7 | 0 | 0.08 | 0 | 0.04 | 0 |
| synthetic $n = 1$ | 0.3 | 35 | 0.32 | 30 | 0.2 | 30 | 0.03 | 1 | 0.1 | 0 | 0.05 | 0 | 0.03 | 0 |
| synthetic $n = 2$ | 0.9 | 160 | 1.03 | 155 | 0.3 | 30 | 0.03 | 1 | 0.1 | 0 | 0.05 | 0 | 0.02 | 0 |
| synthetic $n = 3$ | 2.8 | 785 | 6 | 780 | 0.4 | 30 | 0.04 | 1 | 0.1 | 0 | 0.07 | 0 | 0.02 | 0 |
| synthetic $n = 4$ | 13.5 | 3.9k | 86.3 | 3.9k | 1.2 | 30 | 0.08 | 1 | 0.4 | 0 | 0.14 | 0 | 0.04 | 0 |
| synthetic $n = 5$ | 91.5 | 20k | 2731 | 20k | 17.4 | 30 | 0.14 | 1 | 1.5 | 0 | 0.37 | 0 | 0.09 | 0 |

Table 1 depicts the measured times (Sec) and the number of created elements (Elts) in each scenario. Each created element also represents a deleted element, e.g., through revoking and reapplying a rule or applying a repair rule that creates and deletes elements. In more detail, the table shows measurements for the initial translation of the MoDisco model into the documentation structure and synchronization steps for each scenario using the legacy synchronizer without *repair rules* and the new synchronizer with *repair rules*.

W.r.t. our research questions stated above, we interpret this table as follows: The right columns of the table show clearly that using repair rules preserves all those elements in our scenarios that would otherwise be deleted and recreated by the legacy algorithm[4] **(RQ1)**. The runtime shows a significant performance gain for Scenario 1 including a worst-case model change **(RQ2)**. *Repair rules* do not introduce an overhead compared to the legacy algorithm as can be seen for the synthetic time measurements in Scenario 3 where only one rule application has to be repaired or reapplied. **(RQ2)**. Our new approach excels when the cascade of invalidated rule applications is long. Even if this is not the case, it does not introduce any measurable overhead compared to the legacy algorithm as shown in Scenarios 2 and 3 (**RQ3**).

*Threats to validity.* Our evaluation is based on five real world and five synthetic models. Of course, there exists a wide range of projects that differ significantly from each other due to their size, purpose, and developer styles. Thus, the results may probably differ for other projects. Nonetheless, we argue that the four larger projects extracted from Github are representative since they are part of established tools from the Eclipse community. In this evaluation, we selected three edit operations that are representative w.r.t. their dependency on other

[4] Scenario 1: We expect the new root element to already be translated.

edit operations. They may not be representative w.r.t. other aspects such as size or kind of change, which seems to be of minor importance in this context. Also we limited our evaluation to one TGG rule set due to space issues. However, in our experience the approach shows similar results for a broader range of TGGs which can be accessed through eMoflon.

## 6 Related Work

*Reuse in existing work on TGGs.* Several approaches to model synchronization based on TGGs suffer from the fact that the revocation of a certain rule application triggers the revocation of all dependent rule applications as well [11,18,15]. Especially from a practical point of view such cascades of deletions shall be avoided: In [9], Giese and Hildebrandt propose rules that save nodes instead of deleting and then re-creating them. Their examples can be realized by our construction of *repair rules*. But they do not present a general construction or proof of correctness. This is left as future work in [10] again, where other aspects of [9] are formalized and proven to be correct.

In [3], Blouin et al. added a specially designed repair rule to the rules of their case study to avoid information loss. Greenyer et al. [13] also propose to not directly delete elements but to mark them for deletion and allow for reuse of these marked elements in other rule applications. But this approach comes without any formalization or proof of correctness as well. Again, the given example can be realized as short-cut repair. These uncontrolled and informal approaches are potentially harmful. Re-using elements wrongly may lead to, e.g., containment cycles or unconnected data. Hence, providing precise and sufficient conditions for correct re-use of data is highly desirable as re-use may improve scalability and decrease data-loss. Our short-cut rules formalize when data can be correctly reused. In summary, we do not only offer a unifying principle behind different practically used improvements of TGGs but also give a precise formalization that allows for automatic construction of the rules needed. Thereby, we present conditions under which rule applications lead to valid outputs.

*Comparison to other bx approaches.* Anjorin et al. [2] compared three state-of-the-art bx tools, namely eMoflon [20] (rule-based), mediniQVT [1] (constraint-based) and BiGUL [16] (bx programming language) w.r.t. model synchronization. They point out that synchronization with eMoflon is faster than with both other tools as the runtime of these tools correlates with the overall model size while the runtime of eMoflon correlates with the size of the changes done by edit operations. Furthermore, eMoflon was the only tool able to solve all but one synchronization scenario. One scenario was not solved because it deleted more model elements than absolutely necessary in that case. Using short-cut repair rules, we can solve the remaining scenario and moreover, can further increase eMoflons model synchronization performance.

*Change-preserving model repair.* Change-preserving model repair as presented in [24,21] is closely related to our approach. Assuming a set of consistency-preserving rules and a set of edit rules to be given, each edit rule is accompanied

by one or more repair rules completing the edit step, if possible. Such a complement rule is considered as repair rule of an edit rule w.r.t. an overarching consistency-preserving rule. Operationalized TGG rules fit into that approach but provide more structure: As graphs and rules are structured in triples, a source rule is also an edit rule being complemented by a forward rule. In contrast to that approach, source and forward rules can be automatically deduced from a given TGG rule. By our use of short-cut rules we introduce a pre-processing step to first enlarge the sets of consistency-preserving rules and edit rules.

*Generalization of correspondence relation.* Golas et al. provide a formalization of TGGs in [12] which allows to generalize correspondence relations between source and target graphs as well. They use special typings for the source, target, and correspondence parts of a TGG and for edges between a correspondence part and source and target part instead of using graph morphisms. That approach also allows for partial correspondence relations. But it makes the deletion of elements more complex as it becomes important how many incident edges a node has (at least in the double-pushout approach). We therefore opted for introducing triple graphs with partial morphisms. They allow us to just delete a node without caring if it is needed within an existing correspondence relation.

## 7 Conclusion

Model synchronization, i.e., the task of restoring consistency between two models after a model change, poses challenges to modern bx approaches and tools: We expect them to synchronize changes without losing data in the process, thus, preserving information and furthermore, we expect them to show a reasonable performance. While Triple Graph Grammars (TGGs) provide the means to perform model synchronization tasks in general, both requirements cannot always be fulfilled since basic TGG rules do not define the adequate means to support intermediate model editing. Therefore, we propose additional edit operations being short-cut rules, a special form of generalized TGG rules that allow to take back one edit action and to perform an alternative one. In our evaluation, we show that operationalized short-cut rules allow for a model synchronization with considerably decreased data loss and improved runtime.

To better cope with practical application scenarios, we like to extend our approach by formally incorporating type inheritance, application conditions and attributes in the model synchronization process. Since all of these have been formalized in the setting of ($\mathcal{M}$-)adhesive categories and our present work uses that framework as well, these extensions are prepared but up to future work. Propagating changes from one domain to another is basically done here by operationalizing short-cut rules. A more challenging task is what we call model integration where related pairs of models are edited concurrently and have to be synchronized. These model edits may be in conflict across model boundaries. It is up to future work to allow short-cut rules in model integration. Our hope is to decrease data loss and to improve runtime of model integration tasks as well.

## References


1. Ikv++: Medini QVT. http://projects.ikv.de/qvt
2. Anjorin, A., Diskin, Z., Jouault, F., Ko, H., Leblebici, E., Westfechtel, B.: Benchmarx reloaded: A practical benchmark framework for bidirectional transformations. In: Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017. pp. 15–30 (2017), http://ceur-ws.org/Vol-1827/paper6.pdf
3. Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., Diguet, J.P.: Synchronization of models of rich languages with triple graph grammars: An experience report. In: Di Ruscio, D., Varró, D. (eds.) Theory and Practice of Model Transformations. pp. 106–121. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_8
4. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: A model driven reverse engineering framework. Information and Software Technology **56**(8), 1012–1032 (2014). https://doi.org/https://doi.org/10.1016/j.infsof.2014.04.007
5. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: On principles of least change and least surprise for bidirectional transformations. Journal of Object Technology **16**(1), 3:1–31 (Feb 2017). https://doi.org/10.5381/jot.2017.16.1.a3
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
7. Eppinger, S.D.: Model-based approaches to managing concurrent engineering. Journal of Engineering Design **2**(4), 283–290 (1991). https://doi.org/10.1080/09544829108901686
8. Fritsche, L., Kosiol, J., Schürr, A., Taentzer, G.: Short-Cut Rules. Sequential Composition of Rules Avoiding Unnecessary Deletions. In: Mazzara, M., Ober, I., Salaün, G. (eds.) Software Technologies: Applications and Foundations. pp. 415–430. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_30
9. Giese, H., Hildebrandt, S.: Efficient model synchronization of large-scale models. Tech. Rep. 28, Hasso-Plattner-Institut (2009)
10. Giese, H., Hildebrandt, S., Lambers, L.: Bridging the gap between formal semantics and implementation of triple graph grammars. Software & Systems Modeling **13**(1), 273–299 (Feb 2014). https://doi.org/10.1007/s10270-012-0247-y
11. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software & Systems Modeling **8**(1), 21–43 (Feb 2009). https://doi.org/10.1007/s10270-008-0089-9
12. Golas, U., Lambers, L., Ehrig, H., Giese, H.: Toward bridging the gap between formal foundations and current practice for triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Graph Transformations. pp. 141–155. Springer, Berlin and Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_10
13. Greenyer, J., Pook, S., Rieke, J.: Preventing information loss in incremental model synchronization by reusing elements. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) Modelling Foundations and Applications. Proceedings of the 7th European Conference on Modelling Foundations and Applications. pp. 144–159. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21470-7_11

14. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: 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. pp. 22–31. MDI '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1866272.1866277

15. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Software & Systems Modeling **14**(1), 241–269 (Feb 2015). https://doi.org/10.1007/s10270-012-0309-1

16. Ko, H., Zan, T., Hu, Z.: Bigul: a formally verified core language for putback-based bidirectional programming. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 – 22, 2016. pp. 61–72 (2016). https://doi.org/10.1145/2847538.2847544

17. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. Theoretical Informatics and Applications **39**(3), 511–545 (2005). https://doi.org/10.1051/ita:2005028

18. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Graph Transformations. pp. 401–415. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_27

19. Leblebici, E., Anjorin, A., Fritsche, L., Varró, G., Schürr, A.: Leveraging incremental pattern matching techniques for model synchronisation. In: de Lara, J., Plump, D. (eds.) Graph Transformation. pp. 179–195. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_11

20. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: Di Ruscio, D., Varró, D. (eds.) Theory and Practice of Model Transformations. pp. 138–145. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_10

21. Ohrndorf, M., Pietsch, C., Kelter, U., Kehrer, T.: Revision: a tool for history-based model repair recommendations. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 – June 03, 2018. pp. 105–108. ACM (2018). https://doi.org/10.1145/3183440.3183498

22. Robinson, E., Rosolini, G.: Categories of partial maps. Information and Computation **79**(2), 95–130 (1988). https://doi.org/10.1016/0890-5401(88)90034-X

23. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science, vol. 903, pp. 151–163. Springer (1995). https://doi.org/10.1007/3-540-59071-4_45

24. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair. In: Fundamental Approaches to Software Engineering – 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10202, pp. 283–299. Springer (2017). https://doi.org/10.1007/978-3-662-54494-5_16

## A Additional Preliminaries

In this section, we recall our formal preliminaries thoroughly. We start with the definition and central properties of adhesive categories as introduced by Lack and Sobociński in [17] since they constitute a suitable framework for double-pushout rewriting. Subsequently, we recall rules and the transformations defined by them in adhesive categories and in the category of triple graphs in particular. Moreover, we mention the Local Church-Rosser and the Short-Cut Theorem since we use them to prove Theorem 8.

Adhesive categories can be understood as categories where pushouts along monomorphisms behave like pushouts along injective maps in the category of sets. The definition of an adhesive category uses the notion of van Kampen squares.

**Definition 9 (Van Kampen square and adhesive category).** *A pushout diagram as depicted in Fig. 11 is a* van Kampen square *if for every commutative cube over it (like depicted in Fig. 12) where the backfaces are pullbacks, the front faces are pullbacks iff the top face is a pushout.*

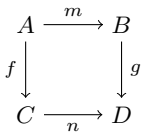



**Fig. 11.** A pushout square

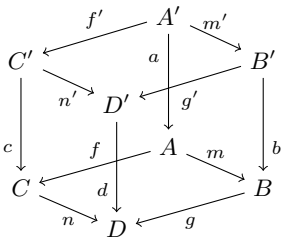



**Fig. 12.** Commutative cube over pushout square

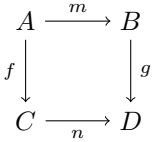
*A category* $\boldsymbol{C}$ *is called* adhesive *if*

1. $\boldsymbol{C}$ *has pushouts along monomorphisms (i.e., pushouts whenever at least one of the two morphisms* $f$ *or* $m$ *in Fig. 11 is a monomorphism),*
2. $\boldsymbol{C}$ *has pullbacks, and*
3. *pushouts along monomorphisms are van Kampen squares.*

Important examples of adhesive categories include the categories of sets, of (typed) graphs, and of (typed) triple graphs [17,6]. We will use the following properties of adhesive categories frequently:

**Fact 10 (Properties of adhesive categories).** *If* $\boldsymbol{C}$ *is an adhesive category, the following properties hold [17]:*

1. *Monomorphisms are stable under pushout, i.e., whenever $m$ (or $f$) is a monomorphism in the pushout diagram to the right, $n$ (or $g$) is a monomorphism. Moreover, pushouts along monomorphisms are pullbacks.*

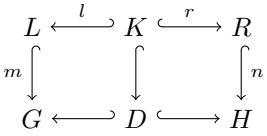


2. *If $f$ is a monomorphism (compare the diagram above), pushout complements for $n \circ f$ are unique (up to isomorphism).*

The definition of rules and their applications is meaningful in arbitrary adhesive categories. Rules offer a declarative means to specify transformations of objects. A rule consists of a left-hand side (LHS) $L$, a right-hand side (RHS) $R$, and a common subobject $K$, called interface. In the context of (triple) graphs, applying a rule intuitively means to delete the elements from $L \setminus K$ and add those of $R \setminus K$.

**Definition 11 (Rules and application).** *Given an adhesive category $\boldsymbol{C}$, a* rule *(or* production*) $p$ consists of three objects $L, K$, and $R$, called* left-hand side*,* interface *(or* gluing object*), and* right-hand side*, and two monomorphisms $l : K \hookrightarrow L, r : K \hookrightarrow R$. Given a rule $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$, the* inverse rule $p^{-1}$ *is defined as $p^{-1} = (R \overset{r}{\hookleftarrow} K \overset{l}{\hookrightarrow} L)$. A rule $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ is called* monotonic *(or* non-deleting*) if $l : K \hookrightarrow L$ is an isomorphism. In that case we just write $r : L \hookrightarrow R$. A* subrule $r'$ *of a monotonic rule $r : L \hookrightarrow R$ is a monotonic rule $r' : L' \hookrightarrow R'$ with monomorphisms $u : L' \hookrightarrow L$ and $v : R' \hookrightarrow R$ such the arising square commutes, i.e., $v \circ r' = r \circ u$. A* common kernel rule *$k$ for monotonic rules $r$ and $r'$ is a common subrule of both.*

*Given a rule $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$, an object $G$, and a monomorphism $m : L \hookrightarrow G$, called* match*, a* (direct) transformation *$G \Rightarrow_{p,m} H$ from $G$ to $H$ via $p$ at match $m$ is given by the diagram to the right where both squares are pushouts. The morphism $n$ is called* co-match *of the transformation, the arising object $D$ its* context object*.*



*A rule $p$ is called* applicable *at match $m$ if the first pushout square above exists, i.e., if $m \circ l$ has a pushout complement.*

*A* grammar *$GG = (\mathcal{R}, S)$ consists of a set of rules $\mathcal{R}$ and a start object $S$. The* language *$\mathcal{L}(GG)$ defined by a grammar consists of all objects $H$ that are derivable by finite sequences of applications of rules of $\mathcal{R}$ starting at $S$, i.e., of objects $S \Rightarrow^*_{\mathcal{R}} H$.*

**Definition 12 (Triple graphs and triple graph morphisms).** *A* graph *$G = (V, E, s, t)$ consists of a set $V$ of vertices, a set $E$ of edges and source and target functions $s, t : E \to V$. A* graph morphism *$f : G \to H$ consists of two functions $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ that are compatible with the assignment of source and target to edges, i.e., $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$. A graph morphism $f = (f_V, f_E)$ is injective/surjective/bijective if both $f_V$ and $f_E$ are.*

*A* triple graph $G = (G_S \xleftarrow{\sigma_G} G_C \xrightarrow{\tau_G} G_T)$ *consists of three graphs* $G_S, G_C, G_T$*, called* source, correspondence*, and* target graph*, and two graph morphisms* $\sigma_G : G_C \to G_S$ *and* $\tau_G : G_C \to G_T$*. A* triple graph morphism $f : G \to H$ *between two triple graphs* $G$ *and* $H$ *consists of three graph morphisms* $f_S : G_S \to H_S, f_C : G_C \to H_C$ *and* $f_T : G_T \to H_T$ *such that* $\sigma_H \circ f_C = f_S \circ \sigma_G$ *and* $\tau_H \circ f_C = f_T \circ \tau_G$*. A triple graph morphism* $f = (f_S, f_C, f_T)$ *is injective/surjective/bijective if* $f_S, f_C$ *and* $f_T$ *all are.*

In practical applications, one assumes the triple graphs to be typed, i.e., instead of considering the category of triple graphs one works in the slice category over a fixed type triple graph $TG$. Practically, this just means that nodes and edges of the graphs get equipped with fixed types that are to be respected by morphisms. The resulting category is an adhesive category with injective, type-preserving triple morphisms as monomorphisms [6]. Adhesive categories have been introduced as a suitable unifying framework for double-pushout rewriting [17] and being adhesive makes the following definitions – not explicitly formulated for (typed) triple graphs – applicable to these, nonetheless.

The following fact is a part of the Short-Cut Theorem [8, Theorem 7] stating the possibilities to analyze the application of a short-cut rule into the sequential application of two rules under certain conditions.

**Fact 13 ((Conditional) Analyzability of short-cut rules).** *In an adhesive category* $\boldsymbol{C}$*, let* $r_i : L_i \hookrightarrow R_i$*,* $i = 1, 2$*, be two monotonic rules,* $k : L_\cap \hookrightarrow R_\cap$ *a common kernel rule for them, and* $r_1^{-1} \ltimes_k r_2$ *the corresponding short-cut rule. Given a direct transformation* $G_1 \Rightarrow_{r_1^{-1} \ltimes_k r_2, m_1'} G_2$ *with context object* $G'$ *such that a pushout complement for* $m_1 \circ r_1 : L_1 \hookrightarrow G_1$ *exists, where* $m_1 = m_1' \circ j_{R_1}$*, then there exists a transformation sequence* $G_1 \Rightarrow_{r_1^{-1}, m_1} G \Rightarrow_{r_2, m_2} G_2$ *compatible with* $k$*. Moreover, a monomorphism* $g : G \hookrightarrow G'$ *exists.*

The Local Church-Rosser Theorem [6, Theorem 5.12] states that sequentially independent transformations may be applied in arbitrary order. In particular, it ensures that in the above definition of sequential independence from a sequence of rule applications the arising transformations $G_i \Rightarrow_{r, e_i \circ d_2^i} G'_{i+1}$ exist in the first place, i.e., that rule $p$ is applicable at the matches $e_i \circ d_2^i$. We recall only a simplified version of one direction of the result, omitting the case for *parallel independence*.

**Fact 14 (Local Church-Rosser Theorem).** *In an adhesive category* $\mathcal{C}$*, given two sequentially independent transformations* $G \Rightarrow_{p_1, m_1} H_1 \Rightarrow_{p_2, m_2'} G'$*, there exist an object* $H_2$ *and transformations* $G \Rightarrow_{p_2, m_2} H_2 \Rightarrow_{p_1, m_1'} G'$*.*

## B Proofs

In this section, we present the proofs of the different statements of the paper.

*Proof (of Lemma 6).* Since the category of triple graphs is adhesive, and the morphism $f_S : D_S \hookrightarrow G_S$ arises as a pushout of the monomorphism $r_S : K_S \hookrightarrow R_S$, it is a monomorphism as well. Moreover, the category of triple graphs has pullbacks and pullbacks of monos are mono, thus $p_G$ is mono and the span $G_C \hookleftarrow G'_C \to H_S$ with $g_S \circ p_D : G'_C \to H_S$ defines a partial morphism. □

*Proof (of Theorem 7).* The transformation $G_S \Rightarrow_{p_S^{\mathrm{pr}}, m_S} H_S$ as depicted in the left part of Fig. 13 (most morphism names are omitted to retain readability) exists by assumption. The partial morphism $G_C \dashrightarrow H_S$ is induced as described by Lemma 6. The pushout complements $D_X$ for the morphisms $m_X \circ l_X$ with $X \in \{C, T\}$ exist by assumption as well as the morphisms $\tau_D : D_C \to D_T$ and $\sigma'_D : D_C \to H_S$; the pushout complement for $n_S \circ id_{R_S}$ is $H_S$ again.

We need to show that there is a morphism $\sigma_D : D_C \to D_S$ such that $k_S \circ \sigma_K = \sigma_D \circ k_C$. If this is the case, the triple $D_S \xleftarrow{\sigma_D} D_C \xrightarrow{\tau_D} D_T$ is the unique pushout complement for $m \circ l$ in the category of triple graphs with $m = (m_S, m_C, m_T)$ and $l = (l_S, l_C, l_T)$. Hence, $G \Rightarrow_{p,m} H$.

By assumption, $H_S D_C D_T \hookrightarrow H_S G_C G_T$ and $R_S K_C K_T \hookrightarrow H_S D_C D_T$ are triple morphisms. Hence, the whole cube in the center of Fig. 13 (the one with the partial morphisms) commutes. First, by definition of commutativity of the bottom square, there exists a morphism $\iota : D_C \hookrightarrow G'_C$ such that

$$f_C = p_G \circ \iota \tag{1}$$

$$g_S \circ p_D \circ \iota = id_{H_S} \circ \sigma'_D \ ; \tag{2}$$

in particular, $\iota$ is mono. We use this morphism to define $\sigma_D \coloneqq p_D \circ \iota$. Moreover, commutativity of the left side square implies the existence of a morphism $x : K_C \hookrightarrow G'_C$ such that

$$p_G \circ x = m_C \circ l_C \tag{3}$$

$$g_S \circ p_D \circ x = n_S \circ r_S \circ \sigma_K \ ; \tag{4}$$

in particular, $x$ is mono. We first calculate that

$$\begin{aligned} p_G \circ \iota \circ k_C &\overset{(1)}{=} f_C \circ k_C \\ &= m_C \circ l_C \\ &\overset{(3)}{=} p_G \circ x \end{aligned}$$

which implies

$$x = \iota \circ k_C \tag{5}$$

(since $p_G$ is mono) and use this to calculate

$$\begin{aligned} g_S \circ \sigma_D \circ k_C &= g_S \circ p_D \circ \iota \circ k_C \\ &\overset{(5)}{=} g_S \circ p_D \circ x \\ &\overset{(4)}{=} n_S \circ r_S \circ \sigma_K \\ &= g_S \circ k_S \circ \sigma_K \end{aligned}$$

which implies the desired result $\sigma_D \circ k_C = k_S \circ \sigma_K$ (since $g_S$ is mono). □

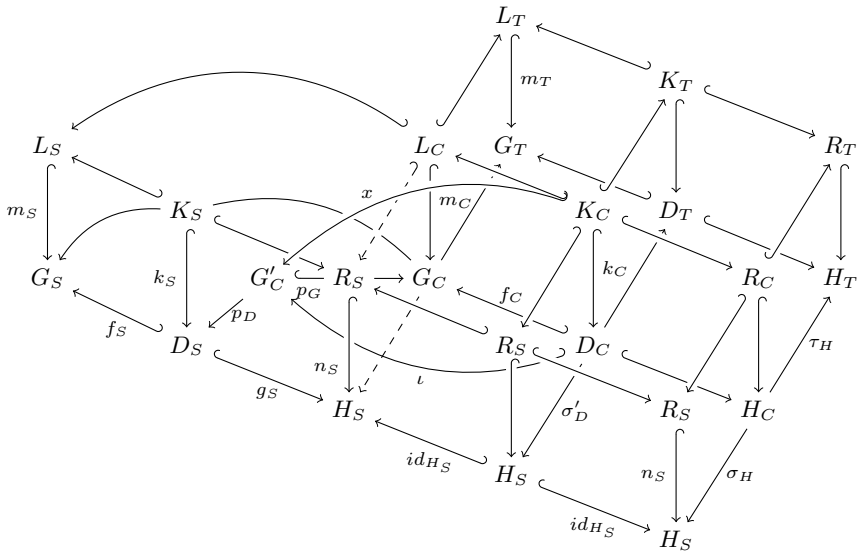



**Fig. 13.** Application of source rule followed by application of forward rule

*Proof (of Theorem 8).* First, because of the sequential independence, repeated application of the Local Church-Rosser Theorem [6, Theorem 5.12] is possible and allows to convert the sequence

$$G \Rightarrow_{r,m} G_0 \Rightarrow_{p_1,m_1} G_1 \Rightarrow_{p_2,m_2} \cdots \Rightarrow_{p_t,m_t} G_t \Rightarrow_{r^{-1} \ltimes_k r', m_{sc}} H$$

into a sequence

$$G \Rightarrow_{r,m} G_0 \Rightarrow_{r^{-1} \ltimes_k r', m'_{sc}} G'_1 \Rightarrow_{p_1,m'_1} \ldots G'_{t-1} \Rightarrow_{p_t,m'_t} H$$

by iteratively switching the order of rule applications.

Secondly, by assumption $m'_{sc} \circ j_R = n$ and furthermore an application of a transformation is invertible [6, Remark 5.3]. Moreover, the transformation $G_0 \Rightarrow_{r^{-1} \ltimes_k r', m'_{sc}} G'_1$ splits into two transformations $G_0 \Rightarrow_{r^{-1},n} G \Rightarrow_{r',m'} G'_1$ by the analysis case of the Short-cut Theorem [8]. Summarizing, we get

$$\begin{aligned} & G \Rightarrow_{r,m} G_0 \Rightarrow_{r^{-1} \ltimes_k r', m'_{sc}} G'_1 \Rightarrow_{p_1,m'_1} \ldots G'_{t-1} \Rightarrow_{p_t,m'_t} H \\ = \quad & G \Rightarrow_{r,m} G_0 \Rightarrow_{r^{-1},n} G \Rightarrow_{r',m'} G'_1 \Rightarrow_{p_1,m'_1} \ldots G'_{t-1} \Rightarrow_{p_t,m'_t} H \\ = \quad & G \Rightarrow_{r',m'} G'_1 \Rightarrow_{p_1,m'_1} \ldots G'_{t-1} \Rightarrow_{p_t,m'_t} H \ . \end{aligned}$$

If $G \in \mathcal{L}(GG)$ for some grammar $GG = (\mathcal{R}, S)$ with $r, r', p_1, \ldots, p_t \in \mathcal{R}$, there exists a finite sequence of transformations $S \Rightarrow^*_{\mathcal{R}} G$ that extends to a finite

sequence $S \Rightarrow^*_{\mathcal{R}} H$ by concatenating with the sequence above. Thus, $H \in \mathcal{L}(GG)$. □

## C Evaluation Ruleset

In this section, we present additional information related to our evaluation from Sect. 5.

Fig. 14 depicts the full TGG rule set used of our evaluation. The first rule *JavaModel-2-DocModel-Rule* defines consistency between a MoDisco *Model* and a *DocModel* that contains three sub *DocModel* and another *Folder* linked to the common *DocModel*. These different containers are used to separate Java entities on the documentation site to split them up into common Java data types, external Java references and source references. *JavaModel-2-DocModel-Rule* then defines consistency between *Packages* and *Folders* given that their parent are a MoDisco *Model* and a *DocModel*, respectively. Using *JavaPackage-2-DocFolder-Rule*, we can now create *Package* and *Folder* hierarchies recursively. Furthermore, there are four rules that define consistency for *ClassesDeclarations*, *Interfaces-Declaration*, *EnumDeclaration* and inner *ClassesDeclarations* each with a *Doc-File*. Also, for the nine primitive types, e.g., boolean, byte and short, consistency is defined between each of them and a *Doc-File*. Given a *ClassDeclaration* or an *InterfaceDeclaration* with its corresponding *Doc-File*, we also define consistency between *MethodDeclarations* on one and *MethodEntries* on the other side. Using the consistency between methods on both sides, we are able to define consistency between *TypeAccesses* and *Parameters*, once for method signatures and once for the return statement. Finally, we define consistency between generalization and realization relationships using three rules. First, a rule for *ClassesDeclarations* that extend another *ClassDeclaration*, second a rule for *InterfacesDeclaration* extending another *InterfaceDeclaration* and last for *ClassesDeclarations* implementing an *InterfaceDeclaration*.

## D Exemplary Repair Rule with Partial Morphism

As was explained in Sect. 4.1, a short-cut rule can be operationalized such that a source edit rule and a corresponding repair rule can be created. However, it was also stated that these source edits may cause the result to no longer be a graph triple because source elements are deleted without deleting their preimage in the correspondence graph. We want to give an example here based on the TGG rule set used in our evaluation and shown in Appendix C. We use *TypeAccess-2-Generalization-Rule* and *ClassDec-2-DocFile-Rule* where the first defines consistency for standard Java inheritance[5] and the latter solely for single *ClassesDeclarations*. In case that we want to revoke the generalization without touching the *ClassDeclaration*, we would like to apply *ClassDec-2-DocFile-Rule* instead.

[5] Note that *ClassDeclaration* is also created here which implicitly forbids multi inheritance

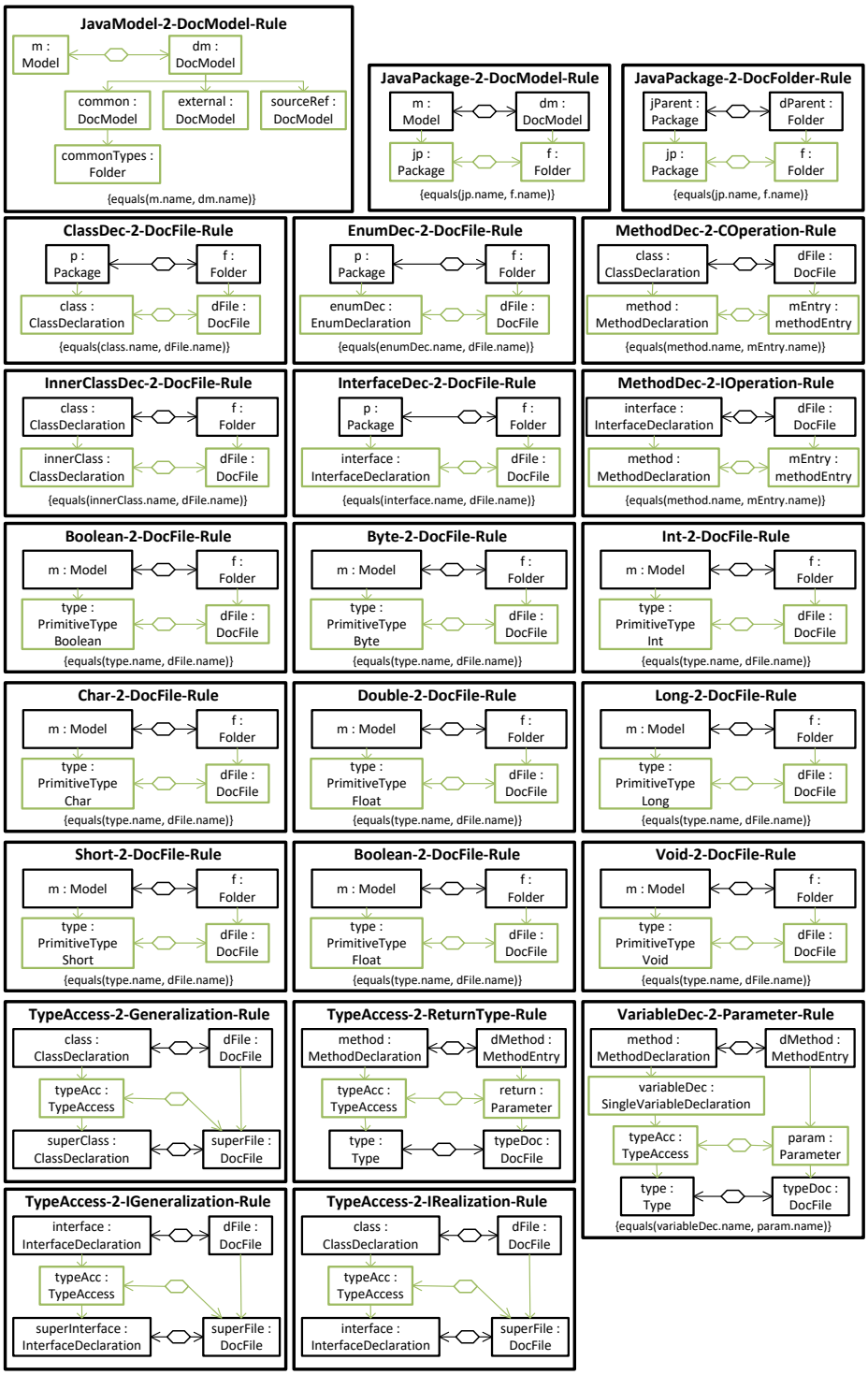



**Fig. 14.** Evaluation - TGG Rule Set

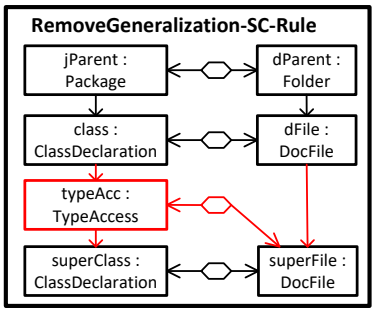



**Fig. 15.** Short-cut Rule

Figure 15 depicts a short-cut rule which can be applied in order to transform a rule application of *TypeAccess-2-Generalization-Rule* into that of *ClassDec-2-DocFile-Rule*. On the source side, we delete the *TypeAccess* element together with all adjacent links plus the correspondence link between *TypeAccess* and `superFile`. Finally, we delete the link between `dFile` and `superFile` which results in deleting all green elements of *TypeAccess-2-Generalization-Rule* that are not present in *ClassDec-2-DocFile-Rule*.

The forward operationalization of this short-cut rule results in the repair rule depicted in Fig. 17. As before, deleted elements on the source side are removed because we expect these elements to already been deleted by the source edit rule which is depicted in Fig. 16. The dashed elements are not part of the source edit rule but depict the current state of the triple at this location. Thus, all elements on correspondence and target side are dashed black and are not altered with except for the edge that connects the correspondence node with `typeAcc`. It is not directly deleted by the source edit rule but deleting `typeAcc` will also lead to this edge being removed. The result is of course no triple graph any more

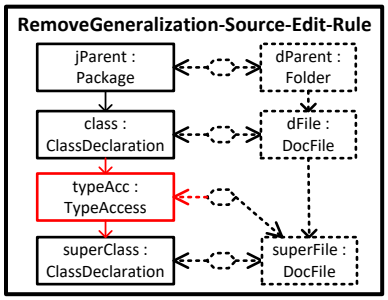



**Fig. 16.** Source Edit Rule

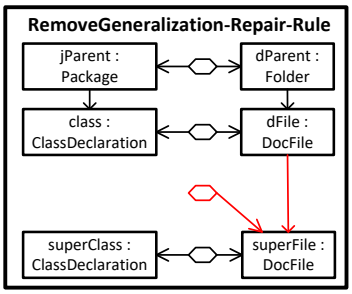



**Fig. 17.** Repair Rule

which has to be fixed by our repair rule from Fig. 17. This also explains the source dangling correspondence which is deleted by the repair rule and results

again in a triple graph. Finally, the rule deletes the reference between `dFile` and `superFile` and restores the consistency of the whole triple with respect to the TGG rule set.