Change-Preserving Model Repair

Gabriele Taentzer[1(✉)], Manuel Ohrndorf[2], Yngve Lamo[3], and Adrian Rutle[3]

[1] Philipps-Universität Marburg, Marburg, Germany
taentzer@informatik.uni-marburg.de
[2] Universität Siegen, Siegen, Germany
mohrndorf@informatik.uni-siegen.de
[3] Western Norway University of Applied Sciences, Bergen, Norway
{Yngve.Lamo,Adrian.Rutle}@hvl.no

Abstract. During modeling activities, inconsistencies can easily occur due to misunderstandings, lack of information or simply mistakes. In this paper, we focus on model inconsistencies that occur due to model editing and cause violation of the meta-model conformance. Although temporarily accepting inconsistencies helps to keep progress, inconsistencies have to be resolved finally. One form of resolution is model repair. Assuming that model changes are state-based, (potentially) performed edit operations can be automatically identified from state differences and further analyzed. As a result, inconsistent changes may be identified causing a need to repair the model. There may exist an overwhelming number of possible repair actions that restore consistency. The edit history may help to identify the relevant repairs. Model inconsistencies are repaired by computing and applying complement edit operations that are needed to re-establish the overall model consistency. In this paper, we clarify under which conditions this kind of model repair can be applied. The soundness of this approach is shown by formalizing it based on the theory of graph transformation. A prototype tool based on the Eclipse Modeling Framework and Henshin is used to conduct an initial evaluation.

Keywords: Model-based engineering · Model repair · Graph transformation

1 Introduction

Model-based engineering has gained increasing popularity in various disciplines, especially in software development. This means that modeling plays a primary role throughout the engineering process and thus, it has to be well supported. While models are edited, they may get inconsistent for various reasons as, e.g., misunderstandings, lack of information, incomplete modeling actions or simply mistakes. Another source of inconsistency may be different interpretations of requirements especially where models are developed collaboratively [1]. In this paper, we focus on model inconsistencies related to the violation of conformance to the underlying meta-model, especially as they occur during editing processes.

Prior to model repair, model inconsistencies have to be detected. Currently many approaches are available that detect inconsistencies fast and correctly, e.g. [2–4]. This may be performed in a check-only mode or integrated with model repair as done in various rule-based approaches such as [5–9].

While it is important to allow inconsistencies during modeling processes [10], they must be resolved eventually. One way of inconsistency resolution is model repair. There are various approaches to repair models which capture this problem in different ways. An overview is given in [11], also stating a common problem in model repair: "One of the main challenges of model repair is that for any given set of inconsistencies, there (possibly) exists an overwhelming number of repair updates that restore the consistency. Yet, since the selection of the most suitable repair is ultimately a choice of the developer, approaches to model repair must balance the automation level of the technique and the need for user guidance in the generation of the repairs." Roughly, we distinguish between the following approaches to model repair: Given an inconsistent model, search-based approaches return a repaired model which is consistent, such as [12–15]. Syntactic and rule-based approaches return a (partially ordered) set of possible repair actions instead (see e.g. repair plans in [8]). Badger [16] is the only model repair approach so far that uses the change history: It uses the date of model revisions to select repair plans but does not consider the performed edit operations to select repair actions.

So far, model repair approaches have not taken the history of user actions into account, hence they miss an important information source for repair generation. By the performed edit operations, users state how they want to evolve the model. Guiding the repair process by the change history can help to identify promising repair possibilities from the overwhelming number of possible ones.

Model changes are either recorded in a state-based manner where just pre- and post-states are stored, or in a delta-based manner where information about the performed user actions is stored. Delta-based approaches have the advantage of keeping the history of the model evolution. If model changes are given state-based, all interesting information about a possible sequence of user actions is not immediately available but can be automatically determined assuming that all the possible user actions have been specified before [17]. Specifications of edit operations can be computed to a large extent as shown in [18]. Hence, independent of the approach that is applied to record changes, we will assume that the delta information can be automatically computed when needed.

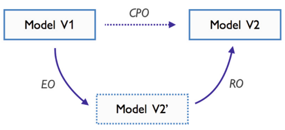

Fig. 1. Change-preserving model repair

Our approach is *rule-based* in the sense that an *edit operation* (EO) is specified by one or more rules depending on its complexity. We assume that a repair action is also specified by a rule, called *repair operation* (RO), such that the composition of an edit operation with a suitable repair operation leads to a *consistency-preserving operation* (CPO), i.e., a rule that – being applied to a consistent model – yields a consistent model again (see Fig. 1). Hence, ROs are complementing preceding

EOs and therefore, preserve already performed model changes. In general, there may be several possible ROs for one EO.

This repair approach is *interactive* since the user may select among several applicable repair actions to repair a model step-by-step. Moreover, it is *consistency improving* with each application of a repair action. We will show as a main result that, if all the repair operations are causally independent from the subsequent edit operations, all inconsistencies can be resolved without side-effects in the sense that no new inconsistencies are introduced while repairing the model, i.e., the repair is *fully consistent* in that case. In addition, if an edit operation does not cause any inconsistency, there is not any repair action to be performed, hence, our approach is *stable*. Our model repairs are not necessarily *least change*, i.e., repaired models are not necessarily as close as possible to the original model, since this is dependent on the specified CPOs.

Typical application scenarios are the following ones: If a model consists of several viewpoints as, e.g., UML models which consist of several diagrams, edit operations in one viewpoint may have to be complemented by actions in other diagrams. If one or several of these complements are forgotten in the original editing process, they may be easily repaired by our approach. Although this complementation might sound a bit mechanical, it cannot be automated in general since often necessary information is lacking. When, e.g., a new method call is inserted in a sequence diagram, it has to be complemented with the definition of a corresponding method in the class model. While the method name may already be fixed by the call, its return type as well as its parameters (with name and type) still have to be specified.

Considering a scenario with just one viewpoint, our approach may also be useful to easily complete more complex edit activities that are not available as separate edit operations. For example, changes of interfaces have to be repeated on implementing classes or attribute changes have to be repeated on their getter and setter methods. Another motivation may be fast editing where just key information is given which can be automatically completed to a bigger extent.

The main contributions of this paper are:

1. A new process for repairing model inconsistencies taking the history of edit operations into account.
2. A formalization of this process using algebraic graph transformation (see e.g. [19]), i.e., a precise formulation of each of its tasks. The main theorem shows that all repair processes of our approach are consistency-improving.
3. A prototypical implementation in Henshin [20], a model transformation language based on the Eclipse Modeling Framework and graph transformation concepts. The prototype is used to conduct a case study where edit processes in the bCMS (Barbados Crash Management System) [21] are considered.

The rest of this paper is organized as follows: The running example is presented in Sect. 2. Section 3 introduces our model repair process. The formalization of our approach follows in Sect. 4. Tool support and an initial evaluation are presented in Sect. 5. Finally, we compare with related work and conclude in Sects. 6 and 7.

2 Running Example

Let us consider an example: A simple class model of an online shop focusing on the customer's viewpoint covers (at least) orders and a shopping cart collecting these orders (see Fig. 2).

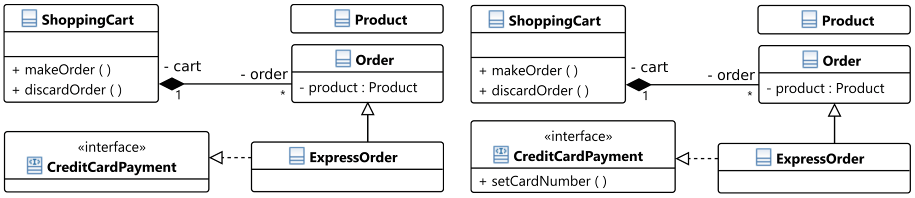

Fig. 2. Class model of an online shop - Versions 0 and 1

The model declares ExpressOrders as a special kind of orders. This class shall implement the interface class CreditCardPayment. The interface is extended by a method called setCardNumber() as shown in Fig. 2. The modeler has introduced an inconsistency here since the class ExpressOrder does not implement this method; the model has to be repaired. A suitable model repair that takes the performed model change into account is to execute the indicated completion, i.e., to add the method also to the class ExpressOrder as shown in Fig. 3.

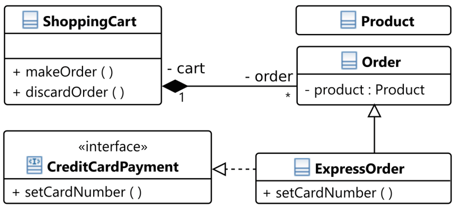

Fig. 3. Class model of an online shop - Inconsistency repaired, Version V1-R

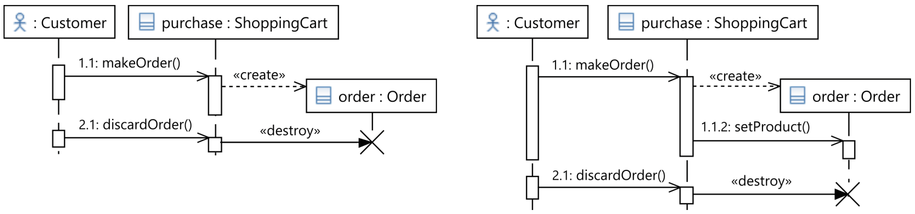

Fig. 4. A sequence diagram for ordering a product - Versions 1 and 2

Building on our example, we will now specify the behavior related to the main use case which is ordering of a product. It is modeled by a simple sequence diagram as shown on the left of Fig. 4 which – together with the class model in Fig. 3 – constitutes a consistent view on the system model in the sense that all object types and messages used are defined in the class model. (Note that attribute values and links are not visible in sequence diagrams.) On the right of Fig. 4 the sequence diagram has been further developed by inserting a new method call of setProduct on the object of type Order. This method is not defined in the current class model; hence, this view of the system model is inconsistent and has to be repaired.

In principle, there are many different repair actions possible, e.g., adding a method setProduct to class Order, removing the message from the sequence diagram while keeping the lifeline of type Order, or removing also this structure. Since the modeler added the message, its removal is a possible repair action. If the modeler selects it, our approach is not specifically helpful for this repair but could be used to identify repairs that are still missing thereafter. But if the modeler wants to keep the added message, our repair approach would immediately propose the missing complement operation which is the addition of the method to the class model; this is shown in Fig. 5.

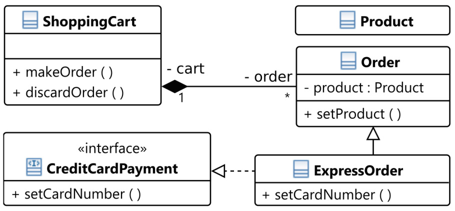

Fig. 5. A class model of an online shop - Inconsistency repaired, version V2-R

3 Model Repair Approach

In the following, we informally present our approach to change-preserving model repair. The approach relies on edit operations (EOs), consistency-preserving operations (CPOs) and repair operations (ROs); EOs and CPOs have to be defined first by the language designer. Thereafter, modelers can repair their inconsistent models.

3.1 Preparing Change-Preserving Model Repair

Before being able to perform change-preserving model repair, the necessary operations for the modeling language and its model editor have to be specified. An overview on the preparation tasks is given in Fig. 6.

For a given modeling language which is specified by a meta-model MM, a set of CPOs w.r.t. the MM has to be defined. These CPOs are usually defined by language designers in cooperation with domain experts. For identifying reasonable edit operations (EOs), we do not consider the original language meta-model but an *effective meta-model* being the original one in a relaxed form, i.e., without (most of the) OCL constraints and with relaxed multiplicities. The effective meta-model defines the language of all possible inputs to a model editor (which may cause inconsistencies w.r.t. the original meta-model). In [18], an automated approach for deriving EOs from effective meta-models is presented and evaluated at several modelling languages and editors.

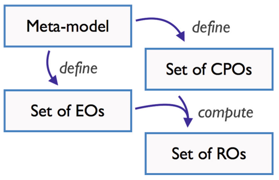

Fig. 6. Preparation for change-preserving model repair

For each CPO, one need to identify sub-operations that are reasonable edit operations in the modeling language, e.g. EOs for inserting, deleting, and moving model parts as well as for changing their attributes [18,22]. An EO is completed to a CPO by a repair operation (RO); i.e., applying an EO followed by an RO to a model, yields the same result as applying the corresponding CPO. In this way, an EO can be seen as a sub-operation of a CPO. It may also happen that an EO is not a sub-operation of any CPO or of several CPOs; in the former case the application of an EO may lead to irreparable inconsistencies while in the latter case, the complement ROs w.r.t. each containing CPO are computed. We stick to the case where each EO is complemented by at most one RO to each of its CPOs. A more general case would be EOs being complemented by sequences of ROs, e.g. adding an interface operation is complemented with adding an operation in each realizing class. In the special case that an EO is already a CPO, model repair is obviously not needed.

3.2 Change-Preserving Model Repair

A change-preserving model repair takes the preceding applications of EOs, i.e., *edit steps*, into account; edit steps that may cause inconsistencies are followed by applications of ROs, called *repair steps*, that re-establish consistency (see Fig. 1). To find out which ROs shall be applied, the edit steps since the last consistent model version are needed. If edit steps are not already provided by the model editor, they can be automatically computed as follows: Given two model versions M_1 and M_2 and a set of EOs, we are looking for a sequence of edit steps from M_1 to M_2. The algorithm presented in [17] yields an *edit script*, i.e., a set of EOs with actual arguments being partially ordered along their sequential dependencies. This algorithm is fast in the sense that it does not need backtracking. It has been implemented for EMF models and applied in several case studies.

Next, the reported edit steps can be analyzed w.r.t. inconsistencies. Each edit step which causes inconsistencies will lead to one of the following two cases:

1. There exists one or more CPOs that have the applied EO as a sub-operation. The modeler chooses one of them and thereby determines the complement RO that has to be performed to re-establish consistency w.r.t. the considered edit step. In Sect. 2 we presented two inconsistent edit steps which are repaired by complements.
2. The edit step introduces a model for which there does not exist any CPO that has the applied EO as a sub-operation. In this case the model change cannot be preserved. It has to be rolled back, at least partly. An example for such an EO (in the context of class models) is the insertion of a generalization relation between two classes such that the overall generalization structure becomes cyclic. The model editor usually allows such an EO.

If all edit steps can be repaired (if needed) and each of the repair steps is causally independent of all the edit steps following the edit step it repairs, we can guarantee the overall consistency of the final model after all repairs (see Theorem 1 below).

The independence of steps can be checked automatically based on the critical pair analysis implemented in Henshin (for more details see Sect. 4.4 below).

4 Formalization

The formalization of the described model repair approach is a means to clarify the assumptions and outcomes of each involved task. Since models can be basically considered as graphs, we rely on the theory of algebraic graph transformation as presented in [19]. In the following, we first recall all basic concepts needed to precisely define models, model changes and modeling languages. We define CPOs as graph transformation rules whose applications preserve the model consistency. Thereafter, change-preserving model repair is formally defined.

4.1 Defining Modeling Languages

When formalizing meta-modeling, graphs occur at two levels: the type level (representing meta-models) and the instance level (given by all valid instance models). This idea is described by the concept of *typed graphs*, where a fixed *type graph* TG together with a set C of constraints serves as an abstract representation of the meta-model. Types are usually structured by an inheritance relation. Multiplicities and other annotations are expressed by additional constraints as well as additional well-formedness rules. The constraints could be defined by means of graph constraints, OCL, first order logic, etc., however the proposed approach is not limited to any particular constraint language. Instances of the type graph are *typed graphs* being equipped with a structure-preserving mapping to the type graph, i.e., a mapping that preserves the source and target functions for edges.

Definition 1 (Meta-model and modeling language). *A meta-model $MM = (TG, C)$ consists of a type graph TG and a set C of constraints typed over TG. All well-typed graphs w.r.t. TG form the set $L(TG)$. All graphs in $L(TG)$ satisfying all the constraints in C form the set $L(MM)$, i.e., the* modeling language *specified by MM.*

Example 1 (UML meta-model excerpt). Looking behind the scenes of our initial example, the example meta-model presented in Fig. 7 is a small and simplified excerpt of the UML meta-model.

It focuses on classifiers with their properties and operations being core ingredients of class models on the one hand and lifelines sending messages as core concepts of sequence diagrams on the other hand. To make the *Realization* relationship between interfaces and classes more precise, we require the following

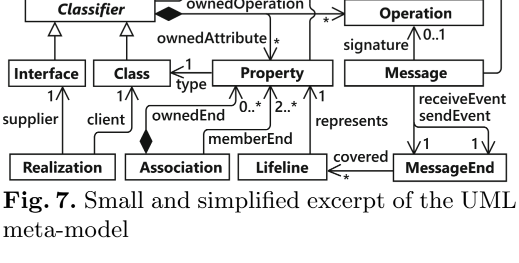

Fig. 7. Small and simplified excerpt of the UML meta-model

constraint: *Each class has to provide all operations which are defined by implemented interfaces.* Another constraint which is used in the running example is this: *Each message has to refer to an operation with the same signature belonging to a class the receiving lifeline is typed over.*

4.2 Model Changes and Their Consistency

Model changes may be formalized by *graph transformation*, i.e. the rule-based modification of graphs. A rule r is defined by two graphs (L, R) and a left application condition AC. L is the left-hand side (LHS) of the rule representing a pattern that has to be found to apply the rule. In addition, this pattern has to fulfill the condition AC before rule application. After the rule application, a pattern equal to R, the right-hand side (RHS), has to occur in the resulting graph. The intersection $L \cap R$, i.e. the graph part that is not changed, and the union $L \cup R$ have to form a graph each. The graph part that is to be deleted is defined by $L \setminus (L \cap R)$, and $R \setminus (L \cap R)$ defines the graph part to be created.

A *graph transformation step* $G \overset{r,m}{\Longrightarrow} H$ between two instance graphs G and H is defined by first finding a match m of the left-hand side L of rule r in the current instance graph G such that m is structure-preserving and type-compatible (i.e., m is a typed graph morphism) and second by constructing H in two passes: (1) building $D := G \setminus m(L \setminus (L \cap R))$, i.e., erasing all the graph items that are to be deleted, and (2) constructing $H := D \cup (R \setminus (L \cap R))$, i.e., adding all the graph items that are to be created. Note that m has to fulfill the *dangling condition*, i.e., all adjacent graph edges of a graph node to be deleted have to be deleted as well, such that D becomes a graph.

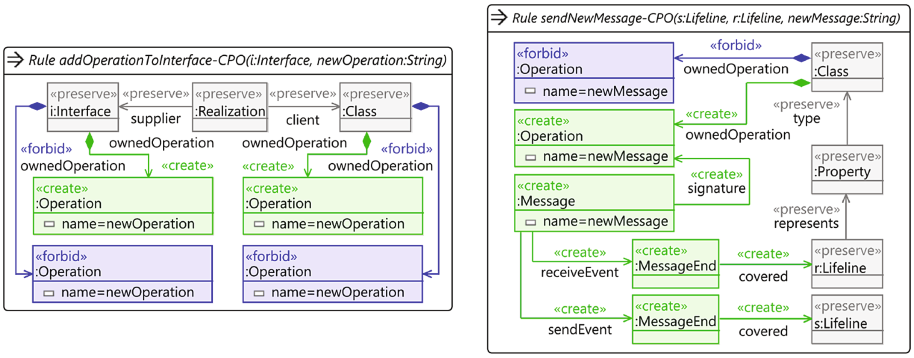

Fig. 8. Rules for adding an operation to an interface class (left) and for sending a new message (right)

Example 1 (Transformation rules). To specify the consistency-preserving operations (CPOs) in our running example, we rely on two rules that preserve the consistency of the simplified UML meta-model. On the left of Fig. 8 the rule for synchronously adding a new operation to an interface and a realizing class is shown. The rule is denoted in a compact form where all elements denoted with preserve are in the LHS, all elements denoted with preserve or create are in the RHS, and all elements denoted with preserve and forbid form a negative application condition. All forbidden elements must not occur in the graph. Note that this rule would have to be extended if there are more than one realizing class for the same interface. Actually a new operation has to be inserted in all the realizing classes. It could also be accomplished with the complement rule in Fig. 10 on the right. The rule on the right of Fig. 8 specifies the synchronous insertion of a message call between two lifelines and its operation into the corresponding class.

Given a UML model in abstract syntax, edit and model repair actions can be expressed by graph transformation steps. Consider, e.g., the graph in Fig. 9 which shows an excerpt of the abstract syntax of the class model on the right of Fig. 2 and the sequence diagram on the left of Fig. 4. Rule *addOperationToInterface* can be applied to the subgraph indicated by V0 and V1 and adds the object marked with V1-R. Applying rule *sendNewMessage* thereafter adds all elements marked with V2 and V2-R as well.

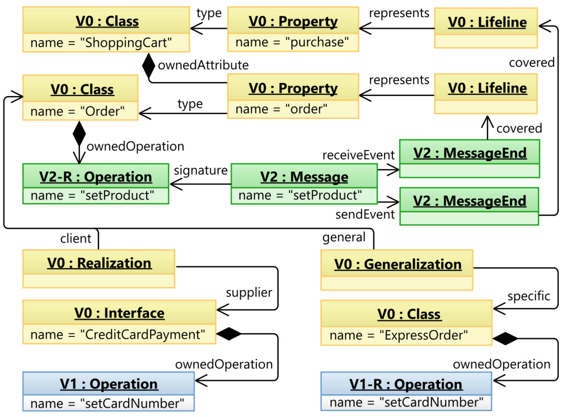

Fig. 9. Example object diagram showing an excerpt of the class model in Fig. 2 and the sequence diagram in Fig. 2 in abstract syntax evolving over time, different versions are indicated by colors and object names

For a given meta-model, all available CPOs can be specified by a graph transformation system.

Definition 2 (Graph transformation system). *Given a set $\mathcal{R}$ of rules, a* graph transformation (sequence) $G \overset{\mathcal{R}}{\Longrightarrow} H$ *consists of zero or more graph transformation steps applying rules of $\mathcal{R}$. A set $\mathcal{R}$ of graph rules, together with a type graph TG, are called a* graph transformation system $GTS = (TG, \mathcal{R})$. *A $GTS = (TG, \mathcal{R})$ is* consistency-preserving *w.r.t. MM if, for every graph G in $L(MM)$, all transformations $G \overset{\mathcal{R}}{\Longrightarrow} H$ yield a consistent graph, i.e., H is in $L(MM)$.*

A modeling language may also be formally defined by a graph grammar GG *being a $GTS = (TG, \mathcal{R})$ together with a* start graph G_0. *It defines a modeling language by all graphs G resulting from transformation sequences starting at G_0, that is, $L(GG) = \{G \in L(TG) | G_0 \overset{\mathcal{R}^*}{\Longrightarrow} G\}$. A graph grammar* conforms *to meta-model MM if $L(GG) \subseteq L(MM)$.*

4.3 Complement Construction

Our model repair approach is mainly based on the complement construction. Given an edit operation (EO), its complement w.r.t. some CPO can be computed. This complement forms the RO to be performed. If an edit operation is already consistency-preserving, it is also a CPO leading to an empty complement rule.

EOs are specified as sub-rules of corresponding CPOs. An EO has to be large enough in the sense that it does not delete nodes that are used as source or target for edges in the super-rule, i.e., it has to fulfill the dangling condition w.r.t. its rule embedding. Otherwise, a complete complement rule cannot be constructed.

Definition 3 (Sub-rule). *A rule* $r_s = ((L_s, R_s), AC_s)$ *is a sub-rule of rule* $r = ((L, R), AC)$ *if* $L_s \subseteq L$, $R_s \subseteq R$, *the inclusion of* r_s *in* r *fulfills the dangling condition, and* AC *can be decomposed into* AC_s *and a (possibly empty) rest application condition.*

Example 2 (Complement rules). A simple EO that just inserts a new operation into an interface has to be complemented by doing so in all realizing classes. This means that the computed RO has to be applied as often as possible to repair all the affected classes. The left rule in Fig. 10 shows the edit rule while the right one shows the corresponding repair rule computed as its complement w.r.t. the left rule in Fig. 8.

The left rule in Fig. 11 inserts a new message between two lifelines and ignores the existence of the corresponding operation in the class model. Since it does not delete anything, it trivially fulfills the dangling condition. The right rule in Fig. 11 shows its complement rule w.r.t. to the overall rule shown on the right of Fig. 8. It inserts the missing operation and its relation to the message call.

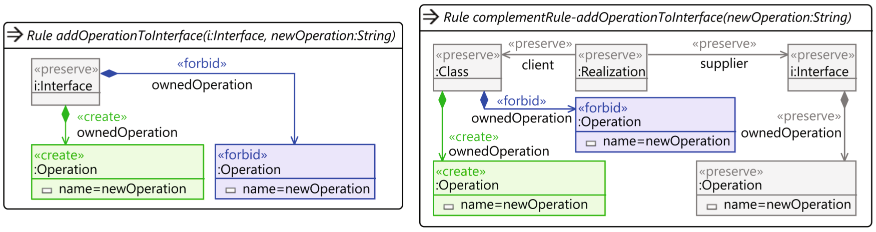

Fig. 10. Adding an operation to an interface: sub-rule (left) and complement rule (right)

Theorem 4.4 in [23] shows that, given a rule r with a sub-rule r_s, there is a canonical way to construct a rule $\bar{r}_s$ with an overlap graph E such that the sequential composition $r_s *_E \bar{r}_s = r$. Such a constructed rule $\bar{r}_s$ is called *complement rule* of r_s w.r.t. r. Furthermore, Fact 4.8 in [23] states that any transformation step $G \xRightarrow{r,m} H$ can be decomposed into two steps $G \xRightarrow{r_s,m_s} \bar{G}$ and $\bar{G} \xRightarrow{\bar{r}_s,\bar{m}_s} H$ of edit step and complement step such that m is an extension of m_s and $\bar{r}_s$ is the complement rule of r_s w.r.t. r.

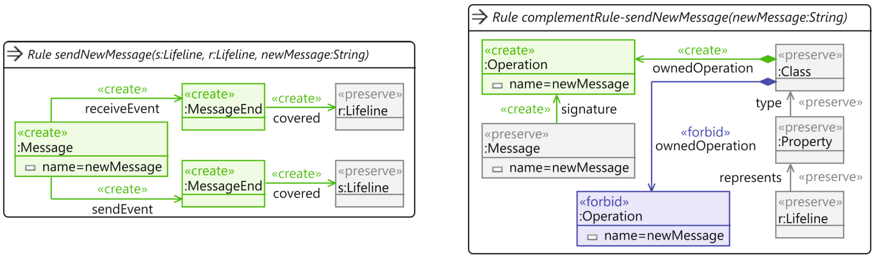

Fig. 11. Sending a message: sub-rule (left) and complement rule (right)

To apply this result to model repair we have to do the following check: Given $G \xRightarrow{r_s,m_s} \bar{G}$ we compute the difference of pre-conditions w.r.t. some $G \xRightarrow{r,m} H$. This means that the match m has to complete the match m_s for $L \setminus L_s$ and all $AC \setminus AC_s$ have to be fulfilled.

If an edit rule is already consistency-preserving, it is a CPO as well. In this case, the complement transformation would apply the empty rule, i.e., $\bar{G} = H$. Hence, our approach is stable in the sense of [11].

4.4 Sequential Independence and Confluence of Transformations

Since the application of complement rules is meant to repair previous edit steps, they usually depend on their edit operations and cannot be applied before the edit step is performed. For example, a graph node has to be created first to further connect it with other model parts. However, the application of a complement rule may be independent of all subsequent edit steps. In that case, a model repair step is called *side effect-free*, and may be exchanged with subsequent edit steps. This means that repairing steps may be performed flexibly throughout the editing process, i.e., immediately after an inconsistent edit step or later – allowing temporary inconsistencies.

Definition 4 (Sequential independence). *Given two transformation steps* $t_1 : G_1 \xRightarrow{r_1,m_1} \bar{G}_2$ *and* $t_2 : \bar{G}_2 \xRightarrow{\bar{r}_2,\bar{m}_2} G_3$*, the execution of* $t_1;t_2$ *is* sequentially independent *if match* m_2 *does not need any element of* $\bar{G}_2$ *newly created by applying* r_1 *and does not use any attribute changed in* t_1*, i.e.,* t_2 *does not need the preceding application of* r_1*. Furthermore, match* m_1 *is not destroyed by* t_2*. The rules* r_1 *and* r_2 *are* sequentially independent *if all sequences* $t_1;t_2$ *applying first rule* r_1 *and then* r_2 *are sequentially independent.*

Example 3 (Independent steps). Considering the edit steps applying first the left rule in Fig. 10 and then the left one in Fig. 11, these two steps are sequentially independent of each other since they do not overlap. Actually, any two applications of these rules are sequentially independent of each other since they act on different viewpoints, i.e., they can never overlap. An execution consisting of an edit step and its repair step by applying the complement rule is usually sequentially dependent. Consider e.g. the rules in Fig. 11 where the message is first inserted and then used to add a corresponding operation.

Checking the sequential independence of rules by hand is tedious. Fortunately this is not necessary since the critical pair analysis (CPA) is a well-known technique to analyze potential conflicts and dependencies of transformation systems. The CPA was originally introduced for term rewriting and later generalized to graph transformation [24]. Henshin contains tool support for the CPA. If there does not exist any critical pair for two given rules r_1 and r_2, all transformation pairs $t_1 : G \overset{r_1,m_1}{\Longrightarrow} H_1$ and $t_2 : G \overset{r_2,m_2}{\Longrightarrow} H_2$ applying these two rules are independent of each other. In this case, the Local-Church-Rosser Property [19] holds ensuring that two independent transformation steps may be executed in any order yielding the same result, i.e., they are also sequentially independent and confluent.

4.5 Change-Preserving Model Repair

The main result in this paper is the following: Given a model change history by a sequence of transformation steps where each rule has at least one complement, each step can be complemented such that a consistent graph can be reached finally. The following result ensures that our approach is fully consistent. To show it we have to ensure that the repair steps are sequentially independent from all edit steps following the edit step it repairs. In that case, repair steps can be arbitrarily interleaved with these edit steps. Several repair steps, however, may dependent on each other such as, for a method call in an interface class, creating first the corresponding interface method and then all corresponding methods in implementing classes.

Theorem 1 (Change-preserving model repair). *Let be given a meta-model MM, a graph transformation system $GTS = (TG, \mathcal{R})$ with a set $\mathcal{R}_s \subseteq \mathcal{R}$ of subrules and $\overline{\mathcal{R}}_s$ of all complement rules of $\mathcal{R}_s$, a graph G in $L(MM)$, and a transformation sequence $G \overset{\mathcal{R}_s}{\Longrightarrow} G_{0n}$. If all rule pairs $(r_{s_i}, \bar{r}_{s_j})$ in $\mathcal{R}_s \times \overline{\mathcal{R}}_s$ for $i > j$ are sequentially independent then there exists a repairing transformation sequence, i.e., a transformation sequence $G_{0n} \overset{\overline{\mathcal{R}}_s}{\Longrightarrow} H$ such that graph H is in $L(MM)$.*

This theorem tells us that a repair is easier if not too many edit steps have to be considered. Its proof can be found in [25]. The main proof idea is to split each CPO into an EO and an RO which can be shifted after all EOs due to the local Church-Rosser property.

Example 4 (Change-preserving model repair). The two subsequent edit steps in our initial example lead both to inconsistencies (of different kinds). Each edit rule is sequentially independent of its opposite repair rule (i.e., the repair rule of the other edit rule) since it just inserts elements that are not needed by the opposite repair rule.

5 Tool Support and Initial Evaluation

Tool support. A first prototype implementation of change-preserving model repair is available at [26]. It is based on the Eclipse Modeling Framework (EMF), Papyrus, and Henshin and supports the following activities: (1) comparison of model versions, (2) recognition of performed edit operations, and (3) provision of concrete repair steps.

Initially the historic version V_1 and the potentially inconsistent version V_2 of a model have to be *compared*. The modeler can choose between different comparison algorithms, e.g. ID-based, signature-based, or similarity-based [27]. Having the set of common model elements available, the tool derives the elementary changes of versions V_1 and V_2, on the level of model elements, references and attributes. We call this kind of change description the technical difference of V_1 and V_2.

For recognizing performed edit operations, the algorithm requires a rule set containing all available EOs for a given model editor. Each edit operation produces a pattern of changes in the technical difference. There is an algorithm presented in [28] which describes how to transform a graph transformation-based rule into a graph pattern which recognizes the corresponding set of changes in the technical difference. In this way, we can recognize all performed EOs. By hiding all those EOs that are already CPOs, only true sub-operations remain in the technical difference showing some inconsistencies. Now the remaining EOs are recognized on the remaining changes using the same algorithm. This even works with incomplete sets of EOs, i.e. elementary changes which cannot be recognized as EOs are ignored.

For each recognized EO we have to *calculate all possible embeddings into corresponding CPOs* (Sect. 4.3). For each pair of CPO and EO rules, a corresponding complement rule RO can be created, by removing all already executed changes of the EO from the CPO.

Next, the *available repair steps are determined*, i.e. the complement rules still have to be provided with parameters for the remaining changes. This means that we have to find all complete matches of the LHS of an RO in the actual model V_2. Usually, a part of the match is already determined by the corresponding EO. Each completed match is reported as a repair to the user. The tool can visualize a selected repair by highlighting the parts in the model diagrams that will be changed. The user can also test a repair by applying and, if necessary, reverting it. The underlying graph transformation logic of the presented approach is transparent to the tool user.

Initial evaluation. For an initial performance estimation, we applied our prototype to the class, sequence and state machine diagrams of the bCMS (Barbados Crash Management System) [21] case study. The UML model V_1 contains approx. 2600 model elements and 22.000 references. The difference to a model V_2 has been computed in $\approx$2 s and contains approx. 600 model elements and 1900 reference changes. 12 inconsistencies were introduced in this edit sequence. Including the EOs of our running example, there are 15 CPOs and 12 EOs. The additional operations consider changes in state machines, e.g. dangling transitions after deleting a state. The EO detection took $\approx$500 ms and filters already

500 consistent changes of 50 steps. (Approx. 2000 changes are ignored since they are not covered by the EOs provided.) The subsequent EO recognition took ≈100 ms; it detects 60 changes performed by 12 steps. After the calculation of the $EO \times CPO$ embeddings in ≈500 ms, a total of 33 concrete repairs were found. The initial performance estimation already allows some conclusions to be made. The performance of the model difference calculation depends on the size of models. The time for the repair calculation, however, mainly depends on the size of the calculated difference and the number of inconsistencies.

To support larger rule sets efficiently, we intend to store rule embeddings in a database. Furthermore, an incremental difference calculation and CPO detection would be interesting for an *online* editing scenario, i.e. suggesting repairs during the user edits the model.

6 Related Work

Most of the existing model repair techniques can be categorized into syntactic, rule- and search-based approaches [11]. In this section, we will compare our approach to existing techniques of those categories.

Syntactic and rule-based approaches. A syntactic repair generator derives repair plans by analyzing the consistency rules at specification time. In rule-based approaches a repair tool is configured with a set of repair rules for frequently occurring inconsistencies. Repair rules or plans are suggested and instantiated at modeling time when inconsistencies occur.

Our approach is closely related to triple graph grammars (TGGs) [29] in the sense that the consistency between two kinds of graphs is specified by rules, called triple rules. Those rules can be used to evolve both related graphs simultaneously in a consistent manner. If the source graph is changed independently by source rules, the corresponding forward rules can be used to synchronize the target graph. A corresponding result is shown in [30]. In this paper, we consider a more general setting where graphs do not have to be sub-structured, edit operations may contain add and delete actions, an edit rule may have several (or no) repair rules (and CPOs), and complex application conditions may be used.

In a wider context, syntactic and rule-based approaches as presented in e.g. [5–9, 31] are interactive and incremental since each inconsistency is repaired separately. While providing full control over the repair process, it may happen, however, that the repair of existing inconsistencies leads to new ones. Criteria for side-effect-free repair are usually not considered. Moreover, the change history is not taken into account to restrict the usually overwhelming number of possible repair rules.

Search-based approaches. Search-based approaches such as [14–16] take the inconsistent model as input and search for a consistent one which is usually related to the original one by least changes. This problem is mostly solved by using a constraint solver. Supporting tools may be parameterizable through the definition of valid edit operations giving the user the possibility to find favored solutions. Current approaches, however, are somewhat restricted to handle small models only; otherwise the search space has to be restricted to user-defined upper bounds w.r.t. instance size, back-tracking or time.

A search-based tool which considers the model history is Badger [16]. The regression planning algorithm can take a variety of parameters to guide the search process. One of them is to prioritize model elements by their version in the model history, e.g., repairs should change preferably newer model elements and keep the older ones. Therefore, the algorithm respects the temporal dimension of the history but does not necessarily preserve the edit operations that have taken place.

7 Conclusion

To handle model inconsistencies, there are often many possible repair actions to consider. In this paper, we have proposed a change-preserving model repair approach to tackle this challenge. Based on the edit operations already performed, model repairs are proposed building on the assumption that the latest change of the model is the most significant. To use this approach, we have to do the following: For a given modeling language we specify a set of consistency-preserving operations. For a given model editor we identify the set of allowed edit operations. The basic idea is to identify the history of performed edit operations since the last consistent model state, and to analyze these operations for identifying those causing inconsistencies. If an inconsistent edit operation turns out to be a sub-rule of a consistency-preserving operation, the complement operation is constructed which is able to repair this inconsistency. The soundness of our approach is shown based on the theory of graph transformation. A prototypical implementation illustrates its practical applicability.

Our approach may be combined with other rule-based approaches: Since rule-based approaches usually allow to choose among a large number of repair operations, their selection needs more guidance. Preserving already performed changes seems to be a reasonable criterion. Our approach supports to find out if a repair operation is a complement of an edit operation such that their sequential combination leads to a consistency-preserving operation. Furthermore, a change-preserving model repair process is side-effect-free if each edit operation is independent of all repair operations of later applied edit operations. It is up to future work to further investigate relations between existing rule-based approaches in different editing scenarios and thereby, integrate change-preservation with other important aspects of model repair.

References

1. Easterbrook, S., Nuseibeh, B.: Using viewpoints for inconsistency management. Softw. Eng. J. **11**(1), 31–43 (1996)
2. Grundy, J.C., Hosking, J.G., Mugridge, W.B.: Inconsistency management for multiple-view software development environments. IEEE Trans. Softw. Eng. **24**(11), 960–981 (1998)

3. Egyed, A.: Instant consistency checking for the UML. In: 28th International Conference on Software Engineering (ICSE), pp. 381–390. ACM (2006)
4. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: 30th International Conference on Software Engineering (ICSE), pp. 511–520. ACM (2008)
5. Enders, B., Heverhagen, T., Goedicke, M., Tröpfner, P., Tracht, R.: Towards an integration of different specification methods by using the viewpoint framework. Trans. SDPS **6**(2), 1–23 (2002)
6. Amelunxen, C., Legros, E., Schürr, A., Stürmer, I.: Checking and enforcement of modeling guidelines with graph transformations. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 313–328. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89020-1_22
7. Königs, A., Schürr, A.: MDI: a rule-based multi-document and tool integration approach. Softw. Syst. Model. **5**(4), 349–368 (2006)
8. Reder, A., Egyed, A.: Computing repair trees for resolving inconsistencies in design models. In: International Conference on Automated Software Engineering, pp. 220–229. ACM (2012)
9. Straeten, R.V.D., D'Hondt, M.: Model refactorings through rule-based inconsistency resolution. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 1210–1217. ACM (2006)
10. Balzer, R.: Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering, pp. 158–165. IEEE Computer Society/ACM Press (1991)
11. Macedo, N., Tiago, J., Cunha, A.: A feature-based classification of model repair approaches. CoRR, vol. abs/1504.03947 (2015)
12. Straeten, R.V.D., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45221-8_28
13. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the kodkod model finder for resolving model inconsistencies. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 69–84. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21470-7_6
14. Sen, S., Baudry, B., Precup, D.: Partial model completion in model driven engineering using constraint logic programming. In: 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) (2007)
15. Macedo, N., Guimarães, T., Cunha, A.: Model repair and transformation with echo. In: 28th International Conference on Automated Software Engineering, ASE 2013, pp. 694–697. IEEE (2013)
16. Puissant, J.P., Straeten, R.V.D., Mens, T.: Resolving model inconsistencies using automated regression planning. Softw. Syst. Model. **14**(1), 461–481 (2015)
17. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 191–201. IEEE (2013)
18. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 173–188. Springer, Cham (2016). doi:10.1007/978-3-319-42064-6_12

19. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006)
20. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16145-2_9
21. Capozucca, A., Cheng, B., Guelfi, N., Istoan, P.: Oo-spl modelling of the focused case study. In: Comparing Modeling Approaches (CMA) International Workshop affiliated with ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (CMA@ MODELS2011) (2011)
22. Rindt, M., Kehrer, T., Kelter, U.: Automatic generation of consistency-preserving edit operations for MDE tools. In: Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS). CEUR Workshop Proceedings, vol. 1255 (2014)
23. Golas, U., Habel, A., Ehrig, H.: Multi-amalgamation of rules with application conditions in M-adhesive categories. Math. Struct. Comput. Sci. **24**(4), 68 (2014)
24. Plump, D.: Critical pairs in term graph rewriting. In: Prívara, I., Rovan, B., Ruzička, P. (eds.) MFCS 1994. LNCS, vol. 841, pp. 556–566. Springer, Heidelberg (1994). doi:10.1007/3-540-58338-6_102
25. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair: extended version. Philipps-Universität Marburg, Technical report (2017). www.uni-marburg.de/fb12/swt/research/publications
26. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair - tool support and initial evaluation. pi.informatik.uni-siegen.de/projects/SiLift/fase2017/
27. Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M.: Adaptability of model comparison tools. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), Essen, Germany, pp. 306–309. ACM (2012)
28. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, KS, USA, pp. 163–172. IEEE (2011)
29. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). doi:10.1007/3-540-59071-4_45
30. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71289-3_7
31. Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MODELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006). doi:10.1007/11880240_15