Empowering model repair: A rule-based approach to graph repair without side effects

1st Alexander Lauer
Philipps-Universität Marburg
Marburg, Germany
alexander.lauer@uni-marburg.de

2nd Jens Kosiol
Philipps-Universität Marburg
Marburg, Germany
kosiolje@mathematik.uni-marburg.de

3rd Gabriele Taentzer
Philipps-Universität Marburg
Marburg, Germany
taentzer@mathematik.uni-marburg.de

***Abstract*—Working with models can lead to inconsistencies due to erroneous or contradictory actions during concurrent modeling processes. Modern modeling environments typically tolerate inconsistencies and support their detection. However, at a later stage of development, models are expected to be consistent, which means that their inconsistencies should be considered and resolved. The process of resolving model inconsistencies is usually referred to as model repair. Our approach to model repair is semi-automatic in the sense that the system computes appropriate paths for repair and the modeler decides which path to go. What is special about our approach is that the repair process can register every small improvement in the model. This allows the interaction with the user to be optimized, resulting in an approach with a high level of automation on the one hand and flexible configuration options on the other. The approach is able to provide all possible repair plans that do not have side effects, i.e., the computed repair plans do not inadvertently introduce a new inconsistency into the model so that a consistent model cannot be achieved after the repair. Since models often have a graph-like structure, we present our approach to model repair based on graphs. Our approach is completely formal and uses the algebraic graph transformation approach to show its correctness.**

***Index Terms*—model repair, graph repair, consistency constraint, graph transformation, graduated consistency**

I. Introduction

Working with models can lead to inconsistencies due to erroneous or contradictory actions in modeling processes, especially in concurrent processes. Modern modeling environments should tolerate inconsistencies and support their detection [1], [2]. However, at a later stage of development, models are expected to be consistent, which means that their inconsistencies should be detected and resolved. The process of resolving model inconsistencies is usually referred to as *model repair*. There are many different approaches to model repair; the approaches previously examined were compared using a feature-based classification of model repair approaches [3]. That article points out that "most techniques do not provide guarantees regarding the functional semantics of the model repair procedures".

There are rule-based approaches that generate repair rule sequences for each inconsistency of a model such as [4] and the user can choose one of the computed repair plans. If two model repairs are in conflict, the repair of one model inconsistency may lead to a new inconsistency as a side effect so that repairs have to be taken back or even a consistent model cannot be achieved after the repair. It is interesting to determine a class of consistency constraint sets where such side effects cannot occur.

Since models are typically based on a graph structure, graph repair approaches can in principle also be used for model repair. Graph repair approaches usually have the advantage that they are formal and thus precisely defined. There are several approaches to graph repair for first-order graph constraints or certain subsets thereof that are correct, i.e. always return consistent graphs [5]–[11]. The used repair algorithm may be non-terminating if there are graph repairs for several constraints that influence each other. We are interested in finding a characterization of consistency constraint sets such that there is a terminating graph repair algorithm that always returns a consistent graph. Ideally, this algorithm is fast in the sense that it does not need backtracking.

Our approach is fully formal and uses the algebraic graph transformation approach [12], [13]. We use a restricted form of nested graph constraints [14] to specify graph consistency. Constraints of this form use alternating quantifiers and do not contain Boolean operators. Graph updates are rule-based and use the algebraic graph transformation approach. Our approach to graph repair is also rule-based and uses so-called repair rules. If a set of constraints is such that repairing one constraint cannot entail a new violation, the constraint set is *circular conflict-free*. This property can be checked statically for a given set of constraints. We show that a graph can always be repaired such that the resulting graph is fully consistent if the constraint set is circular conflict-free. The repair rules can be automatically computed and selected by the modeler. Our repair approach is efficient in the sense that no backtracking is needed.

The paper is organized as follows: Sections II and III present the formal basis for graphs and graph constraints. Consistency checks are defined in Section IV. Graph updates are formalized as graph transformations in Section V. The proper repair algorithm is presented in Sections VI and VII. The paper concludes with the related work in Section VIII and a conclusion in Section IX.

II. Graphs

Since models generally have an underlying graph structure, we present our approach based on graphs. More precisely, we work with typed graphs as introduced for graph transformation in [12], [13]. The nodes of a graph represent objects that can be attributed with data values, and the edges of a graph represent object references.

Definition 2.1 (Graph [13])**.** A *graph* $G = (G_V, G_E, s_G, t_G)$ consists of a set G_V of nodes, a set G_E of edges and two mappings $s_G\colon G_E \to G_V$ and $t_G\colon G_E \to G_V$ that assign the source and target nodes for each edge of G. If a tuple as above is not given, the set of nodes is denoted by G_V and the set of edges by G_E.

In the following, we will assume that a graph is always finite, i.e., the set of nodes and the set of edges are finite.

Definition 2.2 (Graph morphism [13])**.** Given two graphs $G = (G_V, G_E, s_G, t_G)$ and $H = (H_V, H_E, s_H, t_H)$, a *graph morphism* $f\colon G \to H$ consists of two mappings $f_V\colon G_V \to H_V$ and $f_E\colon G_E \to H_E$ that preserve the source and target functions, 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 is called *injective* (*surjective*) if both mappings f_V and f_E are injective (surjective). We denote an injective morphism by $f\colon G \hookrightarrow H$. An injective graph morphism $f\colon G \hookrightarrow H$ is called *inclusion* if $f_E(e) = e$ for all $e \in G_E$ and $f_V(v) = v$ for all $v \in G_V$.

Definition 2.3 (typed graph and typed graph morphism [13])**.** Given a graph TG, called the *type graph*, a *typed graph* over TG is a tuple $(G, type)$ which consists of a graph G and a graph morphism $type\colon G \to TG$. Given two typed graphs $G = (G', type_1)$ and $H = (H', type_2)$, a *typed graph morphism* $f\colon G \to H$ is a graph morphism $f\colon G' \to H'$ such that

$$type_2 \circ f = type_1.$$

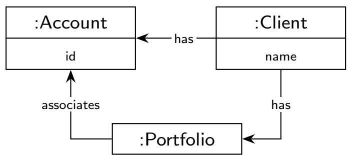


Fig. 1: Type graph

Example 2.1. Throughout this paper, we use a running example from the banking domain. The type graph TG is shown in Fig. 1. It represents the structure of a simple banking example consisting of Clients, Accounts and Portfolios. The edges denoted by has and associates assign an Account or Portfolio to its owner and an Account to the Portfolio it is connected to, respectively. Each Client has an attribute representing its name and each account has an attribute representing its id. The attribute values can be represented by a special kind of node and an edge between an object node and a value node can be interpreted as an attribute. Typed graphs can already cover a simple attribution concept where attributes are only set and not set. For a more powerful concept of attribution that also supports the computation of attribute values, we refer the interested reader to [13] for one possible such formalization.

A graph typed over TG is shown in Fig. 2. The morphism *type* is implicitly given by the type information in the nodes and on the edges.

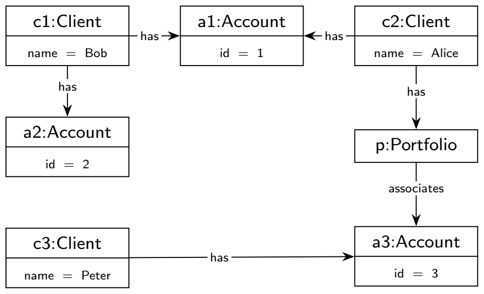


Fig. 2: An instance graph typed over the graph in Fig. 1

III. Constraints

To formulate consistency conditions for graphs, Habel and Pennemann introduced *nested graph conditions* in [14]. The class of nested graph conditions forms a first-order, two-valued logic for graphs [14], [15]. When being used for formalizing invariants, they are called *nested graph constraints*. Radke et al. have shown in [16] that most of OCL (i.e., almost all of the first-order, two-valued part of OCL) can be translated into nested graph constraints.

Since our new repair approach for general nested graph constraints is too extensive to present in this paper, and our approach can well be demonstrated using simple constraints, i.e., constraints with nesting level less than or equal to 2, we will only consider constraints of the following forms in the remainder of this paper:

Definition 3.1 (simple graph constraints (special form of nested graph condition in [14]))**.** A simple graph constraint is one of the following forms:

- $\forall(P, \mathtt{false})$
- $\forall(P, \exists(Q, \mathtt{true}))$ so that there is an inclusion $i_P\colon P \hookrightarrow Q$

The nesting level, denoted by $\mathrm{nl}(c)$, of a simple constraint c is defined as:

- $\mathrm{nl}(\mathtt{true}) = \mathrm{nl}(\mathtt{false}) = 0$.
- $\mathrm{nl}(\forall(P, d)) = \mathrm{nl}(\exists(P, d)) = \mathrm{nl}(d) + 1$.

Note that these forms also cover constraints of the form $\exists(P, \mathtt{true})$, since this constraint can be transformed into the equivalent constraint $\forall(\emptyset, \exists(P, \mathtt{true}))$. The theory for constraints with nesting levels higher than 2 can be found in [17].

Example 3.1. A set of simple constraints is shown in Fig. 3. The inclusions of constraint graphs are not explicitly shown but can be deduced from the node identifiers. For example, the node denoted by c in the first graph of c_1 is mapped to the node denoted with c in the second graph of c_1. Intuitively, the constraints have the following meanings:

- c_1: Each Client has at least one Account.

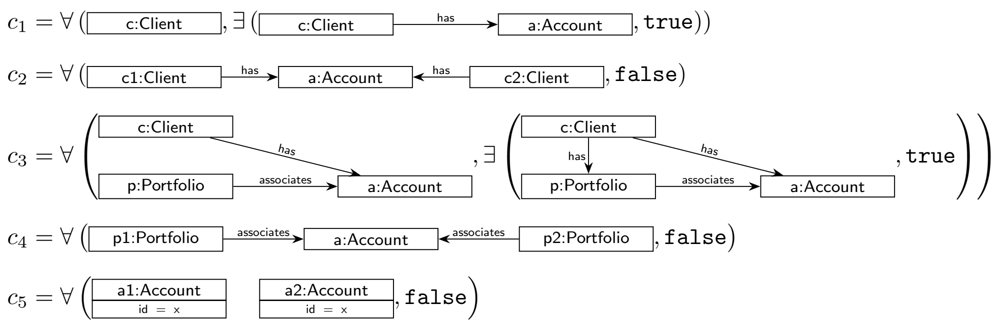

$c_1 = \forall($ c:Client , $\exists($ c:Client has a:Account , true))
$c_2 = \forall($ c1:Client has a:Account has c2:Client , false)
$c_3 = \forall($ c:Client has p:Portfolio associates a:Account , $\exists($ c:Client has p:Portfolio has associates a:Account , true))
$c_4 = \forall($ p1:Portfolio associates a:Account associates p2:Portfolio , false)
$c_5 = \forall($ a1:Account id = x a2:Account id = x , false)

Fig. 3: Constraints used throughout the examples.

- c_2: No Account is assigned to two different Clients.
- c_3: If an Account is associated to a Portfolio, then both belong to the same Client.
- c_4: No Account is contained in two different Portfolios.
- c_5: There are no two different Accounts with the same id, i.e., the id of an Account is unique.

This set of constraints is useful for a simple banking model, and we can think of more constraints, even for this simple model. Another constraint would be, for example, that clients may not have a portfolio without an account.

Let us now introduce the semantics of simple graph constraints and sets of these. For the remainder of this paper we will assume that a set of constraints $\mathcal{C}$ is always finite.

Definition 3.2 (semantic of simple graph constraints)**.** Given a graph G and a simple graph constraint c, then G satisfies c, denoted by $G \models c$, if

- $c = \forall(P, \texttt{false})$ and there is no morphism $p : P \hookrightarrow G$
- $c = \forall(P, \exists(Q, \texttt{true}))$ and for all morphisms $p : P \hookrightarrow G$ there is a morphism $q : Q \hookrightarrow G$ with $p = q \circ i_P$.

A graph G satisfies a set of constraints $\mathcal{C}$ if $G \models c$ for all $c \in \mathcal{C}$.

Example 3.2. Consider the constraints given in Fig. 3 and the graph G given in Fig. 2. The graph satisfies c_1, each Client is connected to an Account. It does not satisfy c_2, the Account with id 1 is assigned to the Clients “Bob” and “Alice”. Also c_3 is not satisfied, Client “Peter” is connected to the Account with id 3 which is connected to a Portfolio, but Client “Peter” is not connected to that Portfolio. Constraints c_4 and c_5 are both satisfied, no Account is associated to two different Portfolios and each Account has a unique id. Therefore, if $\mathcal{C} = \{c_1, c_2, c_3, c_4, c_5\}$, then $G \not\models \mathcal{C}$.

To repair a graph later on, we first search for violating patterns in the graph. Each constraint violation can be identified by the existence of a violating morphism.

Definition 3.3 (violating morphism)**.** Let a graph G and a constraint c be given. If

- $c = \forall(C_1, \texttt{false})$, each morphism $p : C_1 \hookrightarrow G$ is a violating morphism;
- $c = \forall(C_1, \exists(C_2, \texttt{true}))$, each morphism $p : C_1 \hookrightarrow G$ such that $p \not\models \exists(C_2, \texttt{true})$ is a violating morphism.

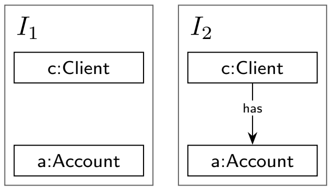


Fig. 4: Intermediate graphs for constraint c_1.

IV. Consistency Checks

To support an appropriate repair of graphs, we will identify a constraint violation as precisely as possible. This means that a constraint is not simply violated or not but can be more or less violated depending on which part of the constraint cannot be satisfied. Thus, we introduce a more fine-grained notion of violation. It allows to detect the degree of consistency of a given constraint in a graph and to identify even the smallest action that leads to an increase in consistency, namely the insertion or deletion of single edges or nodes. A first definition of such a *graduated* notion of consistency of graphs was given in [18]. The following definitions extend the formalization from [18] in such a way that it is not only possible to count violations but also to distinguish their severity.

A prerequisite for the detection of constraint violations is the notion of *intermediate graphs*. A graph C is an intermediate graph of two graphs G and H if it lies in between G and H, i.e., it is a subgraph of H and G is a subgraph of it.

Definition 4.1 (subgraph [19])**.** Given two graphs G and H, G is a *subgraph of* H if there is an inclusion morphism $f : G \hookrightarrow H$. G is called a *proper subgraph of* H if f is not surjective, i.e., $G \neq H$.

Definition 4.2 (intermediate graph)**.** Given two graphs G and H such that G is a proper subgraph of H, a graph G' is called an *intermediate graph of* G *and* H if G is a proper subgraph of G' and G' is a subgraph of H. The set of intermediate graphs between G and H is denoted by $\mathrm{IG}(G, H)$.

Example 4.1. For the constraint c_1 shown in Fig. 3, the intermediate graphs of the first and the second graph of c_1 are shown in Fig. 4.

To decide whether transformations have increased or decreased the consistency level of a constraint, graphs of the constraint are replaced by certain intermediate graphs.

Definition 4.3 (number of violations)**.** Given a graph G and a constraint c, the *number of violations of c in G*, denoted by $\mathrm{nv}(c, G)$, is defined as:

- If $c = \forall(C, \mathtt{false})$:

$$\mathrm{nv}(c, G) = |\{q \mid q\colon C \hookrightarrow G\}|$$

- If $c = \forall(C, \exists(P, \mathtt{true}))$:

$$\mathrm{nv}(c, G) = \sum_{q\colon C \hookrightarrow G} |\{I \in \mathrm{IG}(C, P) \mid q \not\models \exists(I, \mathtt{true})\}|$$

Example 4.2. Consider the graph G given in Fig. 2 and the constraints given in Fig. 3. As discussed in Example 3.2, G satisfies c_1, c_4 and c_5. One sees easily that the satisfaction of a constraint implies that the number of violations is equal to 0 and therefore, $\mathrm{nv}(c_1, G) = \mathrm{nv}(c_4, G) = \mathrm{nv}(c_5, G) = 0$. For c_2, we get $\mathrm{nv}(c_2, G) = 2$, since the Account with id 1 is assigned to more than one Client. In fact, it is associated with exactly two Clients. So there are two morphisms from the graph of c_2 to G. For c_3, there is only one morphism p from the first graph of c_3 to G involving Client "Peter", the Account with id 3 and the Portfolio node. The set of intermediate graphs of c_3 contains only the second graph of c_3. Since p cannot be extended to this intermediate graph, we get $\mathrm{nv}(c_3, G) = 1$.

V. UPDATES

Models can generally be updated by simply changing the state or by applying defined operations. This mainly depends on the type of model editor used. Most diagram editors support basic editing operations where model elements can be created from a palette and relationships between elements are inserted so that no hanging relations can occur. The application of such editing operations can be easily used to derive a delta between subsequent model states. Both types of model update can be formalized on the basis of graphs, and the application of edit operations can be specified using graph transformations.

In the following, we recall the algebraic approach to graph transformation presented in [12], [13].

Definition 5.1 (graph transformation rule [13])**.** A *graph transformation rule* $\rho = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ consists of graphs L, K and R and inclusions $l : K \hookrightarrow L$ and $r : K \hookrightarrow R$.

Example 5.1. The graph transformation rules being used in our example are shown in Fig. 5. They are presented in a compact notation used in Henshin [20], where deleted elements are coloured in red, created elements are coloured in green and preserved elements are coloured in grey (and all elements are additionally stereotyped according to their role). According to Definition 5.1, the left-hand side L contains the elements coloured in red and grey, the context K contains the elements coloured in grey, and the right-hand side R contains all elements coloured in grey and green. These rules can be

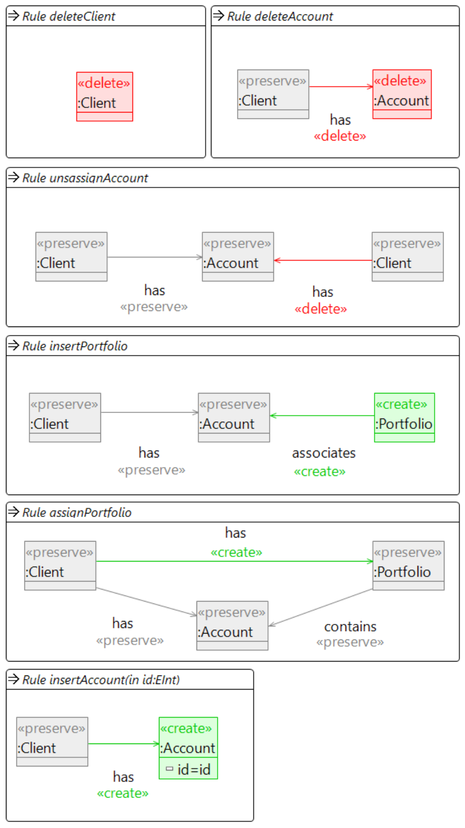


Fig. 5: Graph transformation rules used in the examples.

seen as a specification of selected editing operations for a simple domain-specific modeling language in banking.

In the following, we will recall the definition of a graph transformation. For simplicity, despite the common definition of graph transformations using the double-pushout approach (DPO) based on the concept of pushouts from category theory, we will present a more constructive definition based on set theory, which has been shown to be equivalent to the definition using the DPO when dealing with graphs [13].

When a transformation rule is applied to a graph, all nodes and edges of $L \setminus K$ that match to the graph are deleted, and a copy of all nodes and edges of $R \setminus K$ is added to the graph.

Definition 5.2 (graph transformation [13])**.** Let a graph G, a rule $\rho = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ and an injective graph morphism $m : L \hookrightarrow G$ be given. A *transformation* t, denoted by $t\colon G \Longrightarrow_{\rho,m} H$, via ρ at m can be constructed as follows:

1) Delete all nodes and edges of L that do not have a preimage in K, i.e., construct the graph $D = G \setminus m(L \setminus l(K))$.
2) Add all nodes and elements of R that do not have a preimage in K, i.e., construct the graph $H = D \dot{\cup} R \setminus r(K)$, where $\dot{\cup}$ denotes the disjoint union.

If and only if D is a graph, i.e., it does not contain any dangling edges, ρ is *applicable* at m and m is called *match*. By construction D is a subgraph of both, G and H. Therefore, there are inclusions $g : D \hookrightarrow G$ and $h : D \hookrightarrow H$ which we call *transformation morphisms*.

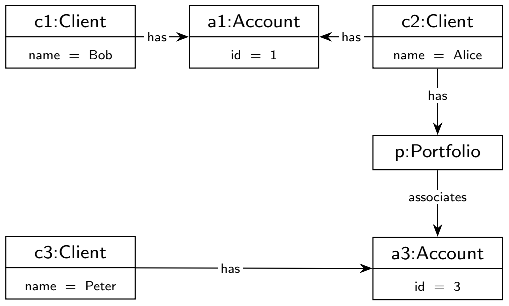


Fig. 6: Graph after applying the rule *deleteAccount* (Fig. 5) to the graph in Fig. 2.

Example 5.2. Consider graph G given in Fig. 2 and the rule *deleteAccount* given in Fig. 5. There is an injective morphism from the left-hand side of *deleteAccount* to G, where the Client node is mapped to Client "Bob", the Account node to the Account with id 2 and edge to the corresponding edge that lies in between them. When *deleteAccount* is applied at this match, we obtain the graph H in Fig. 6.

Rule *deleteAccount* is not applicable at the morphism that maps the Client node to Client "Bob" and the Account node to the Account with id 1. The deletion of the Account would produce a dangling edge of type has originating in Client "Alice".

Graph transformations can, in particular, be used for repairing models. In this case, we are interested in knowing whether the consistency increases continuously in a repair process. In the following, we introduce two special forms of graph transformation that are related to the consistency of a given constraint.

Definition 5.3 (Consistency-increasing and consistency-maintaining transformation)**.** Given a constraint c, a transformation $t : G \Longrightarrow H$ is called *consistency maintaining w.r.t.* c if

$$\mathrm{nv}(c, H) \leq \mathrm{nv}(c, G).$$

The transformation is called *consistency increasing w.r.t.* c if

$$\mathrm{nv}(c, H) < \mathrm{nv}(c, G).$$

These notions of transformation allow new violations to be introduced as long as enough violations are also deleted. In certain cases, it is necessary to ensure that no new violations are introduced at all. To reflect this, we introduce the even stricter notions of *direct consistency-increasing* and *direct consistency-maintaining* transformations.

Definition 5.4 (direct consistency-increasing and -maintaining transformation)**.** Given a transformation $t\colon G \Longrightarrow H$ and a constraint c, the transformation t is called *direct consistency-increasing (direct consistency-maintaining) w.r.t.* c if it is consistency-increasing (consistency-maintaining) w.r.t. c and does not introduce any new violations of c. A rule ρ is *direct consistency-increasing (direct consistency-maintaining) w.r.t.* c if all the transformations applying ρ are.

By definition, a consistency-increasing transformation is also consistency-maintaining; a direct consistency-increasing (direct consistency-maintaining) transformation is also consistency-increasing (consistency maintaining). For a more detailed comparison of these notions to the kinds of transformations introduced in [14] and [18] we refer to [17].

Example 5.3. The transformation $t\colon G \Longrightarrow H$, as described in the Example 5.2, is a direct consistency-maintaining transformation with respect to all constraints. The constraints c_1, c_4 and c_5 are still satisfied in H, $\mathrm{nv}(c_2, H) = \mathrm{nv}(c_3, H) = 1$ and no new violations of c_2 and c_3 have been introduced. A transformation $t'\colon G \Longrightarrow H'$ via the rule *unassignAccount*, which matches to the Clients "Bob" and "Alice" and the Account with id 1 and removes the has-edge between "Alice" and the Account, is direct consistency-increasing with respect to c_2, since $H' \models c_2$.

VI. Repair Preparation

Before we present our graph repair algorithm, we need some prerequisites. These are mainly a characterization of constraint sets that are satisfiable and that can be repaired in a certain order one after the other, so that no repair cycles occur. Such a cycle can occur when the repair of some violations adds new violations to the graph from constraints whose violations have already been repaired.

Definition 6.1 (conflict within a constraint)**.** Let a constraint $c = \forall(P, \exists(Q, \mathtt{true}))$ be given.

1) The graph P *causes a conflict for* Q if a deletion of an occurrence of P can also delete an occurrence of Q without deleting the embedded occurrence of P, i.e., there is an intermediate graph $C \in \mathrm{IG}(\emptyset, P)$ such that the rule $\rho = P \longleftrightarrow C \hookrightarrow C$ is not a direct consistency-maintaining rule w.r.t. $\exists(Q, \mathtt{true})$.
2) The graph Q *causes a conflict for* P if the creation of an occurrence of Q at an already existing occurrence of P can introduce a new occurrence of P, i.e., the rule $P \longleftrightarrow P \hookrightarrow Q$ is not a direct consistency-maintaining rule w.r.t. $\forall(P, \mathtt{false})$.

Definition 6.2 (circular conflict-free constraint)**.** A constraint c is called *circular conflict-free* if

- $c = \forall(P, \mathtt{false})$ or
- $c = \forall(P, \exists(Q, \mathtt{true}))$ and P does not cause a conflict for Q or Q does not cause a conflict for P.

Example 6.1. Consider constraint $c_3 = \forall(C_1, \exists(C_2, \mathtt{true}))$ given in Fig. 3. An insertion of an has-edge between the Client and Portfolio nodes of an occurrence of C_1 will never introduce a new occurrence of C_1. Hence, c_3 is circular conflict-free. In fact, all constraints shown in Fig. 3 are circular conflict-free.

The core concept of our repair algorithm is that of a repair sequence, i.e., a transformation sequence that repairs a constraint violation in a minimal context. In the repair algorithm, rules derived from repair sequences, so-called *derived repair rules* are applied at violating morphisms. For constraints of the form $\forall(C, \texttt{false})$ or $\forall(C, \exists(P, \texttt{true}))$ we are looking for a repair sequence that does not delete any nodes in the occurrence of C. (Please note that edges in the occurrence of C may be deleted.)

We leave open which repair rules to use and where they might come from. In [8]–[11], Habel and Sandmann derive the repair rules from the constraint graphs. By identifying a missing or too large graph part, a rule can be determined that performs exactly the desired action. Another possibility is to consider basic editing operations as they occur in model editors and to specify these operations as rules.

Definition 6.3 (repair sequence)**.** Let a set of rules $\mathcal{R}$ and a condition c over a graph C be given. A transformation sequence

$$C \Longrightarrow_{\rho_1} G_1 \Longrightarrow_{\rho_2} \ldots \Longrightarrow_{\rho_n} G_n$$

via rules $\rho_1, \ldots, \rho_n \in \mathcal{R}$ is a *repair sequence for c via $\mathcal{R}$* if $G_n \models c$ and no node of C is deleted. If the repair sequence does not delete any elements of C, we call it an *inserting sequence* and otherwise a *deleting sequence*.

There are two kinds of repair sequences for a constraint of the form $\forall(C, \exists(P, \texttt{true}))$. One sequence that inserts a new occurrence of P at C, and one that destroys an occurrence of C. Constraints of the form $\forall(C, \texttt{false})$ can only be repaired by destroying occurrences of C. We can do so by deleting edges of the occurrences of C. The requirement that no node of C is deleted ensures that the sequence is applicable to any violating morphism of the constraint, regardless of the context in the graph to be repaired [17].

Example 6.2. Consider constraint c_1 (Fig. 3). One application of the rule *insertAccount* at the first graph of c_1 forms a repair sequence for c_1, since the resulting graph satisfies c_1 and no element of the first graph of c_1 is deleted. An application of the rule *createAccountAndClient* in Fig. 7 forms another repair sequence for c_1. However, this application would introduce a side effect, i.e., a new violation of c_2.

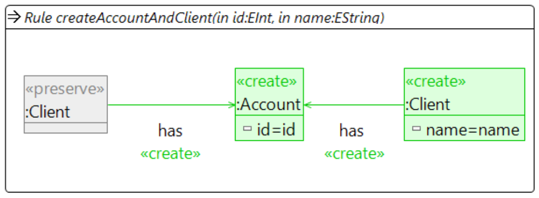


Fig. 7: Rule *createAccountAndClient*.

To determine a suitable rule set for repairing a constraint violation, we check whether there is at least one repair sequence for that constraint via a given set of rules. Then, this rule set is a repair rule set. We have briefly discussed above the kinds of rules that are promising for this task.

Definition 6.4 (repair rule set)**.** Given a set of rules $\mathcal{R}$ and a constraint c, then $\mathcal{R}$ is called a *repair rule set* for c if there is a repair sequence for c via $\mathcal{R}$. $\mathcal{R}$ is a *repair rule set* for a set of constraints $\mathcal{C}$ if $\mathcal{R}$ is a repair rule set for each $c \in \mathcal{C}$.

Given a repair rule set $\mathcal{R}$ for a circular conflict-free constraint c, any graph can be repaired via $\mathcal{R}$ in our approach. However, there may also be cases where $\mathcal{R}$ is not a repair rule set, but G can still be repaired via $\mathcal{R}$.

Example 6.3. The rule set $\mathcal{R} = \{$*insertAccount*, *unassignClient*, *assignPortfolio*$\}$ forms a repair rule set for c_1, c_2 and c_3. For constraints c_4 and c_5, there is no repair rule set using the rules in Fig. 5, since no rule is able to delete an Account that is associated with a Portfolio or an edge between an Account and a Portfolio.

For the definition of conflicts between different constraints, we look for repair sequences that are not consistency-maintaining. To decide this, we summarize a repair sequence in a derived rule [13], here *derived repair rule*. To define a derived repair rule, we use the *track morphism* of a transformation, which allows us to track elements of the original graph into the resulting graph of the transformation [21].

Definition 6.5 (track morphism [21])**.** Given a transformation $t : G \Longrightarrow H$ with the transformation morphisms $g \colon D \hookrightarrow G$ and $h \colon D \hookrightarrow H$, the *track morphism* of t, denoted by $\mathrm{tr}_t : G \dashrightarrow H$, is a partial morphism defined as

$$\mathrm{tr}_t = \begin{cases} h(g^{-1}(e)) & \text{if } e \in g(D) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The construction of *derived repair rules* is a special case of the construction of derived spans, which allows to derive a rule, given a sequence of transformations, which can be used to transform the first graph directly into the last graph of the sequence, i.e., there is a transformation $t : G_1 \Longrightarrow_{\rho} G_n$ where G_1 is the first, G_n the last and ρ the derived rule of the sequence [13].

Definition 6.6 (derived repair rule [13])**.** Let a repair sequence

$$C \overset{t_1}{\Longrightarrow}_{\rho_1} G_1 \overset{t_2}{\Longrightarrow}_{\rho_2} \ldots \overset{t_n}{\Longrightarrow}_{\rho_n} G_n$$

for a constraint c be given. The *derived repair rule* ρ of this sequence is constructed as follows:

1) If the sequence is an inserting sequence,

$$\rho := C \xhookleftarrow{\mathrm{id}} C \xhookrightarrow{\mathrm{tr}_{t_n} \circ \ldots \circ \mathrm{tr}_{t_1}} G_n.$$

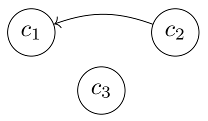
2) If the sequence if a deleting sequence,

$$\rho := C \xhookleftarrow{i_1} C' \xhookrightarrow{i_2} G_n$$

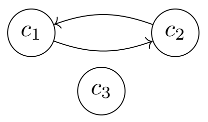
where $C' = \mathrm{tr}_{t_n} \circ \ldots \circ \mathrm{tr}_{t_1}(C)$ and i_1 and i_2 are the inclusions of C' in C and G_n respectively.

As there is usually not only one constraint that can cause violations but a set of several constraints, we also have to consider the interrelationships between repairs of the different constraints, which can also lead to conflict situations.

Definition 6.7 (conflict between constraints)**.** Given a set of constraints $\mathcal{C}$ and a repair rule set $\mathcal{R}$ for $\mathcal{C}$, constraint c *causes a conflict for* constraint c' if there is a repair sequence for c



(a) Conflict graph of $\mathcal{C}$ w.r.t. $\mathcal{R}$.



(b) Conflict graph of $\mathcal{C}$ w.r.t. $\mathcal{R}'$.

Fig. 8: Conflict Graphs.

via $\mathcal{R}$ such that the derived repair rule of this sequence is not a direct consistency-maintaining rule w.r.t. c'.

In order to easily see whether there are circular conflicts between constraints, we use a conflict graph that represents each constraint as a node and each conflict between two constraints as an edge. A circular conflict is then represented as a cycle in the conflict graph.

Definition 6.8 (conflict graph, circular conflict freeness)**.** Given a rule set $\mathcal{R}$ and a constraint set $\mathcal{C}$, the *conflict graph* of $\mathcal{C}$ w.r.t. $\mathcal{R}$ is constructed in the following way: There is a node for each constraint $c \in \mathcal{C}$. If c causes a conflict for c' w.r.t. $\mathcal{R}$, there is an edge e with $s(e) = c$ and $t(e) = c'$.

A constraint c causes a *transitive conflict* for c' w.r.t. $\mathcal{R}$ if the conflict graph contains a path from c to c'. $\mathcal{C}$ is *circular conflict-free* w.r.t. $\mathcal{R}$ if every constraint of $\mathcal{C}$ is circular conflict-free and no constraint of $\mathcal{C}$ causes a transitive conflict for itself w.r.t. $\mathcal{R}$.

In other words, a constraint set $\mathcal{C}$ is circular conflict-free w.r.t. $\mathcal{R}$ if its conflict graph w.r.t. $\mathcal{R}$ is acyclic.

Example 6.4. Consider the constraint set $\mathcal{C} = \{c_1, c_2, c_3\}$ and the set of rules $\mathcal{R} = \{$*unassignAccount*, *insertAccount*, *assignPortfolio*$\}$. There is a conflict from c_2 to c_1. An application of *unassignAccount* also leads to a destruction of an occurrence of the second graph of c_1. There are no further conflicts, so $\mathcal{C}$ is circular conflict-free w.r.t. $\mathcal{R}$. If we consider the rule set $\mathcal{R}' = \{$*unassignAccount*, *createAccountAndClient*, *assignPortfolio*$\}$, there is an additional conflict between c_1 and c_2. An application of *createAccountAndClient* is a repair sequence for c_1 but also introduces a new occurrence of the first graph of c_2. Hence, $\mathcal{C}$ has an circular conflict w.r.t. $\mathcal{R}'$. Both conflict graphs are shown in Figure 8.

Given a set of constraints $\mathcal{C}$ and a set of repair sequences $\mathcal{RS}$ such that this set contains at least one repair sequence for each constraint in $\mathcal{C}$, it suffices to compute the conflict graph representing only the conflicts caused by the repair sequences of $\mathcal{RS}$, provided that only these sequences are used for repair.

Given a set of constraints, there may be no graph that satisfies all the constraints, i.e., the constraints may contradict each other. If a constraint set is circular conflict-free, it is also satisfiable. This is a consequence of the correctness and termination of our repair algorithm (Theorem 7.1).

Definition 6.9 (satisfiable constraint)**.** A simple constraint c is satisfiable if there is a graph G that satisfies c.

Corollary 6.1. A set of constraints $\mathcal{C}$ is satisfiable if there is a repair rule set $\mathcal{R}$ for $\mathcal{C}$ so that $\mathcal{C}$ is circular conflict-free w.r.t. $\mathcal{R}$.

During repair, we use the topological order of the conflict graph to determine the repair order of the constraints. In particular, the use of this order is an essential part of proving the correctness and termination of the algorithm.

Definition 6.10 (topological order [19])**.** Given a graph G, a sequence $(v_1, \ldots, v_n)$ of nodes in G is called a *topological order* of G if no edge $e \in E_G$ with $s(e) = v_i$, $t(e) = v_j$ and $i \geq j$ exists.

VII. Repair algorithm

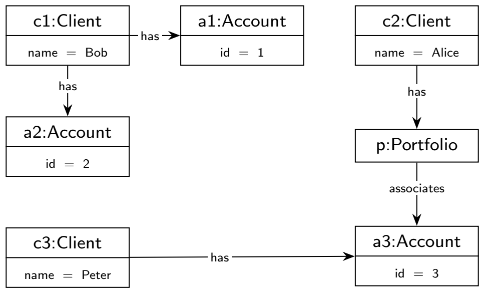
Now we are ready to present our algorithm for graph repair. This algorithm can repair all violations one by one. During the repair, no side effects occur where repairs have to be taken back or even a consistent graph cannot be achieved. Hence, the repair algorithm terminates and all violations can be repaired. The main assumption for this kind of repair is that the set of constraints is circular conflict-free, i.e., the constraints do not influence each other too much. Hence, the constraint violations can be repaired in a topological order so that new violations of already repaired constraints are not introduced.

Our approach is divided into two parts. Algorithm 1 repairs one circular conflict-free constraint. Algorithm 2 uses Algorithm 1 to repair a circular conflict-free set of constraints.

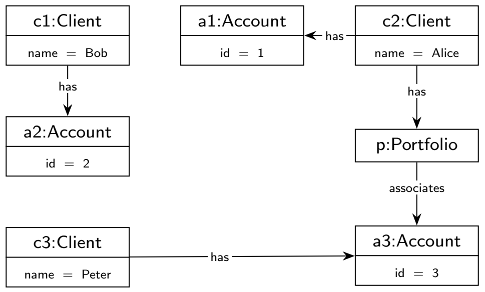
Algorithm 1: Repair algorithm for one circular conflict-free constraint

```
Data: A graph $G$, a circular conflict-free constraint $c$
      over a graph $C$ and a repairing rule set $\mathcal{R}$ for $c$.
Result: A graph $H$ with $H \models c$.
1 Determine a set of repair sequences for $c$;
2 while $G \not\models c$ do
3     Determine the set $P$ of all violating morphisms;
4     Choose $p : C \hookrightarrow G \in P$;
5     Choose a repair sequence for $c$;
6     Apply the repair sequence at match $p$, let $H$ be the
        resulting graph;
7     $G \leftarrow H$;
8 end
9 return $G$;
```

It would be sufficient to compute the set P once. Since one application of a repair sequence can repair several violating morphisms at once, we compute P in each iteration to avoid applying a repair sequence to a morphism that has been repaired in a previous iteration.



(a) Repair by deleting the edge between Client "Alice" and Account a1.



(b) Repair by deleting the edge between Client "Bob" and Account a1.

Fig. 9: Possible results of Algorithm 1.

Example 7.1. Let us assume that the rule set $\mathcal{R}$ contains all the rules shown in Fig. 5. Consider again the constraint c_2 (Fig. 3) and the graph G (Fig. 2). As already described in Example 4.2, $\mathrm{nv}(c_2, G) = 1$. Algorithm 1 can repair c_2 by applying the rule *unassginAccount* in two different ways. Either by removing the edge running from Client "Bob" to the Account with id 1 or by removing the edge running from Client "Alice" to this Account. Both possible outcomes of this repair are shown in Fig. 9. Note that the first repair has the unpleasant side effect that constraint c_1 is no longer satisfied, since Client "Alice" is not associated with any Account.

```
Algorithm 2: Repair algorithm for a finite, circular
conflict-free set of constraints
  Data: A graph $G$, a finite, circular conflict-free set of
        constraints $\mathcal{C}$ and a repairing set $\mathcal{R}$ for $\mathcal{C}$.
  Result: A graph $H$ with $H \models \mathcal{C}$.
1 Choose a topological order $(c_1, \dots, c_n)$ of the conflict
    graph of $\mathcal{C}$ w.r.t. $\mathcal{R}$;
2 for $i \leftarrow 1$ to $n$ do
3     Repair $c_i$ in $G$ with Algorithm 1, let $H$ be the
        resulting graph;
4     $G \leftarrow H$ ;
5 end
6 return $G$;
```

Example 7.2. As discussed in Example 4.2, the graph G in Fig. 2 does not satisfy the set $\mathcal{C} = \{c_1, c_2, c_3\}$ and

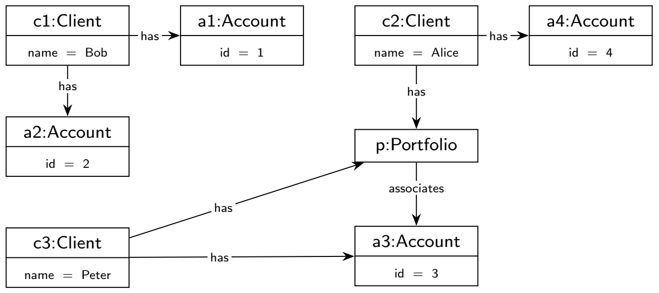


Fig. 10: Repair result using Algorithm 2 with $\mathcal{C} = \{c_1, c_2, c_3\}$ and $\mathcal{R} = \{\mathit{unassignAccount}, \mathit{insertAccount}, \mathit{assignPortfolio}\}$.

$\mathcal{R} = \{\mathit{unassignAccount}, \mathit{insertAccount}, \mathit{assignPortfolio}\}$ is a repairing set for $\mathcal{C}$ (Example 6.4). As the topological order of the conflict graph (Fig. 8a), we can choose either (c_2, c_1, c_3), (c_3, c_2, c_1) or (c_2, c_3, c_1). If (c_2, c_1, c_3) is chosen, Algorithm 2 repairs c_2 in the first step and creates one of the graphs shown in Fig. 9. The graph in Fig. 9a does not satisfy c_1. Therefore, first c_1 and then c_3 are repaired. A possible result of the repair process is shown in Fig. 10.

Theorem 7.1. Given a graph G, a finite, circular conflict-free set of constraints $\mathcal{C}$ and a repair set $\mathcal{R}$ for $\mathcal{C}$. Then, Algorithm 2 is correct, i.e., it terminates and returns a graph that satisfies $\mathcal{C}$.

Proof. Given an topological order $(c_1, \dots, c_n)$ of the conflict graph of a constraint set $\mathcal{C}$ w.r.t. to a repair rule set $\mathcal{R}$ for $\mathcal{C}$.

According to the definition of direct consistency-maintaining rules, a repair sequence for c_i cannot introduce new violating morphisms for a constraint c_j with $j < i$. Since Algorithm 1 only applies derived repair rules, a repair of c_i in a graph that already satisfies c_j will result in a graph that still satisfies c_j. Thus, if the number of constraints is finite and the constraints are repaired according to the topological order, the resulting graph satisfies $\mathcal{C}$.

It remains to show that Algorithm 1 terminates and thus, the resulting graph satisfies constraint c whose violations are to be repaired. By Definition 6.3, an application of a repair sequence for c does not introduce any new violating morphism. Thus, the finite number of violating morphisms decreases with every iteration of the while loop. Therefore, the set P is empty after a finite number of iterations. By the definition of violating morphisms (Definition 3.3), one can easily see that the absence of violating morphisms implies the satisfaction of the constraint. Therefore, Algorithm 1 terminates after a finite number of iterations. □

Corollary 7.1. Algorithm 1 terminates and returns a consistent graph.

Discussion: To better understand how our graph repair algorithm can be used for model repair, we discuss its properties along the feature-based classification of model repair approaches in [3]. Our graph repair algorithm is *semi-automatic*, in that the user can choose a repair sequence for each violation and these repair sequences are then used to repair the graph G. The algorithm is also *incremental*, since it repairs one violation at a time. If there is no constraint violation, nothing happens which means that the algorithm is *stable*. For each graph repair, the user can *configure* the set of constraints whose violations are repaired and also the set of repair rules to be applied. However, the absence of circular conflicts is checked for the selected sets before the actual repair is made. The repairs are not necessarily *least-changing*, as this property depends directly on the repair rules chosen.

VIII. Related Work

In the following, we compare our repair approach with other rule-based approaches to graph and model repair.

A. Rule-based graph repair

There are several approaches to rule-based graph repair [6], [8]–[11], [22], [23]. The approach closest to ours is that of Habel and Sandmann [8]–[11]: Similar to our approach, Habel and Sandmann define a formal approach based on graph transformation and nested graph constraints in alternating normal form (ANF). More precisely, they define rule-based graph repair for constraints in ANF that have a nesting level less than or equal 2 or end with $\exists(C, \mathtt{true})$. They have presented a terminating repair algorithm for this form of constraints that is able to produce consistent graphs. Their algorithm derives rules from the constraints; alternatively, a given rule set (specifying, e.g., basic editing operations for graphs) can be checked for equivalence to the derived rules. Considering repair in terms of a single constraint, our new concept of circular conflict-free constraints (Def. 6.2) is an important step beyond their work. Sandmann and Habel prove termination only for repair programs called *solid* in [11], which are severely restricted in that they are forced to reuse existing graph elements wherever possible. For example, in our running example, once an Account exists, their solid repair program repairs our constraint c_1 by connecting Clients without Accounts to pre-existing Accounts (likely leading to violations of c_2). In terms of repairing a set of constraints, our approach can be seen as complementary to theirs. They are able to repair *preserving sequences*, i.e. sequences of constraints where the repair program does not introduce violations of previously repaired constraints. They show that the property of being preserving is semi-decidable [11, Lemma 22]. Our notion of circular conflict-freeness of a set of constraints (Def. 6.8) can be understood as a new, *statically checkable* sufficient condition for the sortability of a set of conditions into a preserving sequence. Finally, our fine-grained consistency check which is able to detect small improvements (at the level of individual elements) even if full consistency has not yet restored, is a new formal concept.

In [6], the authors formalize rule-based model repair on the basis of graphs and precisely characterize the type of constraints for which the repair algorithm terminates and results in consistent models. However, the type of constraints is quite limited and only includes the constraints for models in the Eclipse Modeling Framework (EMF) and multiplicity constraints. (Note that multiplicity constraints can also be specified with exactly the two forms of simple constraints presented in this paper.)

In [23], the authors consider constraints of the form $\forall(C_1, \exists(C_2, \mathtt{true}))$. They extend given transformation rules to so-called *interaction schemes* [24] such that the creation of new occurrences of C_1 is accompanied by the then necessary creation of an occurrence of C_2. Under certain conditions, this formalism can also be used to repair existing violations of such constraints.

In addition, there is a large amount of work on triple graph grammars (TGGs) [25] where two graphs are interconnected by an intermediate one. In TGGs, updates of one graph are propagated to the other graph via the intermediate graph. In [22], this kind of graph repair is optimized so that the resulting graphs satisfy constraints of the form $\forall(C, \mathtt{false})$.

B. Rule-based model repair

There are also several approaches to rule-based model repair such as [4], [5], [26]–[29]. As a good example for the state-of-the-art of rule-based model repair without formal foundation we discuss a recent work in more detail. In [4], the authors give an up-to-date presentation of a major approach to rule-based model repair. In this approach, abstract repairs are generated for the various causes of inconsistencies. To repair an inconsistency, there may be multiple repair alternatives in form of repair sequences, which are organized in repair trees [27]. This approach has been implemented for UML models and OCL constraints and has been extensively evaluated in practice for larger models and constraint sets. It is shown that this approach is powerful and suitable for practice. Since it is not a formal approach, its correctness has not been proved. Specifically, it does not provide static conditions for conflict-free repair, as our approach does.

IX. Conclusion

Rule-based graph and model repair are very popular because it allows constraint violations to be repaired in a flexible, interactive way. Rule-based repair supports the separate repair of each violation and the flexible selection of a suitable repair alternative. The main disadvantage of rule-based approaches is that side effects can occur in the sense that repairing a violation can inadvertently introduce new violations. In this paper, we have presented a rule-based approach to graph repair so that no backtracking due to side effects is required to repair all violations. This is possible if the set of constraint does not contain circular conflicts within the same constraint or between different constraints. This additional assumption can be checked statically, i.e., directly on the constraint set. It does not require the actual graph with constraint violations.

Nevertheless, circular conflict-free nature of constraints is, of course, a limitation of our repair approach. It is the task of future work to develop repair strategies for constraint sets with circular conflicts. For example, one strategy could be to first identify the largest sets of circular conflict-free constraints, so that the user can select one of these sets and begin a fast repair of all corresponding violations, and then attempt to repair the remaining violations by hand.

Furthermore, we plan to implement our repair algorithm based on Henshin and its conflict analysis, and to apply it to larger case examples. In this way, we would like to evaluate how suitable our graph repair approach is for practical use.

On the theoretical side, we plan to extend our graph repair approach to different variants of graph-like structures, which is possible if the constructions are defined on the basis of category theory, as well as to constraints with deeper nesting levels, as already considered in [17] for typed graphs.

Acknowledgment

We would like to thank Andy Schürr for his valuable comments on an earlier version of this paper.

References

[1] P. Stevens, "Bidirectionally tolerating inconsistency: Partial transformations," in *Fund. Approaches to Software Engineering - 17th Int. Conf., FASE 2014*. Springer, 2014, pp. 32–46. [Online]. Available: https://doi.org/10.1007/978-3-642-54804-8_3

[2] C. Mayr-Dorn, R. Kretschmer, A. Egyed, R. Heradio, and D. Fernández-Amorós, "Inconsistency-tolerating guidance for software engineering processes," in *43rd IEEE/ACM Int. Conf. on Software Eng.: New Ideas and Emerging Results, ICSE (NIER) 2021*. IEEE, 2021, pp. 6–10. [Online]. Available: https://doi.org/10.1109/ICSE-NIER52604.2021.00010

[3] N. Macedo, J. Tiago, and A. Cunha, "A feature-based classification of model repair approaches," *IEEE Trans. Software Eng.*, pp. 615–640, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2620145

[4] L. Marchezan, R. Kretschmer, W. K. G. Assunção, A. Reder, and A. Egyed, "Generating repairs for inconsistent models," *Softw. Syst. Model.*, pp. 297–329, 2023. [Online]. Available: https://doi.org/10.1007/s10270-022-00996-0

[5] N. Nassar, H. Radke, and T. Arendt, "Rule-based repair of EMF models: An automated interactive approach," in *Theory and Practice of Model Trans. - 10th Int. Conf., ICMT@STAF 2017*. Springer, 2017, pp. 171–181. [Online]. Available: https://doi.org/10.1007/978-3-319-61473-1_12

[6] N. Nassar, J. Kosiol, and H. Radke, "Rule-based Repair of EMF Models: Formalization and Correctness Proof," in *Graph Computation Models (GCM 2017), Electronic Pre-Proceedings*, 2017. [Online]. Available: pages.di.unipi.it/corradini/Workshops/GCM2017/papers/Nassar-Kosiol-Radke-GCM2017.pdf

[7] S. Schneider, L. Lambers, and F. Orejas, "A logic-based incremental approach to graph repair," in *Fundamental Approaches to Software Engineering – 22nd Int. Conf., FASE 2019*. Springer, 2019, pp. 151–167. [Online]. Available: https://doi.org/10.1007/978-3-030-16722-6_9

[8] A. Habel and C. Sandmann, "Graph repair by graph programs," in *Software Technologies: Applications and Foundations – STAF 2018*. Springer, 2018, pp. 431–446. [Online]. Available: https://doi.org/10.1007/978-3-030-04771-9_31

[9] C. Sandmann and A. Habel, "Rule-based graph repair," in *Proceedings Tenth Int. Workshop on Graph Computation Models, GCM@STAF 2019*, 2019, pp. 87–104. [Online]. Available: https://doi.org/10.4204/EPTCS.309.5

[10] C. Sandmann, "Graph repair and its application to meta-modeling," in *Proceedings of the Eleventh Int. Workshop on Graph Computation Models, GCM@STAF 2020*, B. Hoffmann and M. Minas, Eds., 2020, pp. 13–34. [Online]. Available: https://doi.org/10.4204/EPTCS.330.2

[11] ——, "A theory on graph generation and graph repair with application to meta-modeling," Ph.D. dissertation, University of Oldenburg, 2021. [Online]. Available: http://uol.de/f/2/dept/informatik/download/Promotionen/Sandmann_Dissertation.pdf

[12] H. Ehrig, "Introduction to the algebraic theory of graph grammars (A survey)," in *Graph-Grammars and Their Application to Computer Science and Biology, Int. Workshop, 1978*. Springer, 1978, pp. 1–69. [Online]. Available: https://doi.org/10.1007/BFb0025714

[13] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. [Online]. Available: https://doi.org/10.1007/3-540-31188-2

[14] A. Habel and K. Pennemann, "Correctness of high-level transformation systems relative to nested conditions," *Math. Struct. Comput. Sci.*, pp. 245–296, 2009. [Online]. Available: https://doi.org/10.1017/S0960129508007202

[15] A. Rensink, "Representing first-order logic using graphs," in *Graph Transformations, Second Int. Conf., ICGT 2004*. Springer, 2004, pp. 319–335. [Online]. Available: https://doi.org/10.1007/978-3-540-30203-2_23

[16] H. Radke, T. Arendt, J. S. Becker, A. Habel, and G. Taentzer, "Translating essential OCL invariants to nested graph constraints for generating instances of meta-models," *Sci. Comput. Program.*, pp. 38–62, 2018. [Online]. Available: https://doi.org/10.1016/j.scico.2017.08.006

[17] A. Lauer, "Rule-based graph repair using minimally restricted consistency-improving transformations," Master's thesis, Philipps-Universität Marburg, Department of Mathematics and Computer Science, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09150

[18] J. Kosiol, D. Strüber, G. Taentzer, and S. Zschaler, "Sustaining and improving graduated graph consistency: A static analysis of graph transformations," *Sci. Comput. Program.*, 2022. [Online]. Available: https://doi.org/10.1016/j.scico.2021.102729

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[20] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place EMF model transformations," in *Model Driven Engineering Languages and Systems - 13th Int. Conf., MODELS 2010*. Springer, 2010, pp. 121–135. [Online]. Available: https://doi.org/10.1007/978-3-642-16145-2_9

[21] D. Plump, "Confluence of graph transformation revisited," in *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*. Springer, 2005, pp. 280–308. [Online]. Available: https://doi.org/10.1007/11601548_16

[22] A. Anjorin, A. Schürr, and G. Taentzer, "Construction of integrity preserving triple graph grammars," in *Graph Trans. - 6th Int. Conf., ICGT 2012*. Springer, 2012, pp. 356–370. [Online]. Available: https://doi.org/10.1007/978-3-642-33654-6_24

[23] J. Kosiol, L. Fritsche, N. Nassar, A. Schürr, and G. Taentzer, "Constructing constraint-preserving interaction schemes in adhesive categories," in *Recent Trends in Algebraic Development Techniques – 24th IFIP WG 1.3 Int. Workshop, WADT 2018*. Springer, 2019, pp. 139–153. [Online]. Available: https://doi.org/10.1007/978-3-030-23220-7_8

[24] U. Golas, A. Habel, and H. Ehrig, "Multi-amalgamation of rules with application conditions in M-adhesive categories," *Math. Struct. Comput. Sci.*, 2014. [Online]. Available: https://doi.org/10.1017/S0960129512000345

[25] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science, 20th Int. Workshop, WG '94*. Springer, 1994, pp. 151–163. [Online]. Available: https://doi.org/10.1007/3-540-59071-4_45

[26] X. Blanc, A. Mougenot, I. Mounier, and T. Mens, "Incremental detection of model inconsistencies based on model operations," in *Advanced Information Systems Engineering, 21st Int. Conf., CAiSE 2009*. Springer, 2009, pp. 32–46. [Online]. Available: https://doi.org/10.1007/978-3-642-02144-2_8

[27] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *IEEE/ACM Int. Conf. on Automated Software Eng., ASE'12*. ACM, 2012, pp. 220–229. [Online]. Available: https://doi.org/10.1145/2351676.2351707

[28] M. Ohrndorf, C. Pietsch, U. Kelter, L. Grunske, and T. Kehrer, "History-based model repair recommendations," *ACM Trans. Softw. Eng. Methodol.*, pp. 15:1–15:46, 2021. [Online]. Available: https://doi.org/10.1145/3419017

[29] A. Barriga, R. Heldal, A. Rutle, and L. Iovino, "PARMOREL: a framework for customizable model repair," *Softw. Syst. Model.*, pp. 1739–1762, 2022. [Online]. Available: https://doi.org/10.1007/s10270-022-01005-0