

Graph Repair and its Application to Meta-Modeling*

Christian Sandmann

Universität Oldenburg
Oldenburg, Germany

`christian.sandmann@uni-oldenburg.de`

Model repair is an essential topic in model-driven engineering. We present typed graph-repair programs for specific conditions; application to any typed graph yields a typed graph satisfying the condition. A model graph based on the Eclipse Modeling Framework (EMF), short EMF-model graph, is a typed graph satisfying some structural EMF-constraints. Application of the results to the EMF-world yields model-repair programs for EMF k constraints, a first-order variant of EMF constraints; application to any typed graph yields an EMF k model graph. From these results, we derive results for EMF model repair.

1 Introduction

In model-driven software engineering, the primary artifacts are models [23, 8]. Models have to be consistent w.r.t. a set of constraints, specified for example in the Object Constraint Language (OCL) [15]. To increase the productivity of software development, it is necessary to automatically detect and resolve inconsistencies arising during the development process (see, e.g. [14, 11, 13]).

To enable automated model repair or model completion, we look for an algorithm that - given a meta-model with two constraints and any model satisfying one of the constraints - creates another model satisfying the old as well as a new one (see Figure 1).

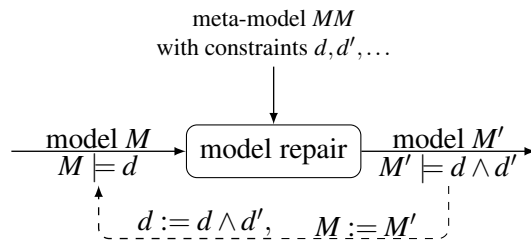


Figure 1: General idea to model repair

If we have such an algorithm, the process can be iterated: Using the model satisfying two constraints, and a new constraint as input, the algorithm creates a model satisfying the conjunction of three constraints, and so on. This iterative approach is necessary in handling large conditions. In each step, one condition is handled. If all steps terminate, and in all steps the preceding conditions remain preserved, we can

*This work is partly supported by the German Research Foundation (DFG), Grants HA 2936/4-2 and TA 2941/3-2 (Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages).

be sure that, after the consideration of all finitely many conditions, the conjunction of the conditions is satisfied. If after one step a preceding condition is violated, the condition must be considered again and the process may be come non-terminating.

In this paper, we represent a meta-model as a type graph, the instance model as a graph typed over the type graph, and first-order constraints as a typed graph constraint, equivalent to a first-order graph formula [20]. Given a typed constraint, we extract a typed program from the constraint, called *repair program*, such that the application of the typed repair program to an arbitrary typed graph yields a typed graph satisfying the constraint.

For small (basic) constraints, we extract a basic repair program directly from the constraint. For larger (proper) constraints, the repair program is composed from basic repair programs. For generalized proper constraints repair programs for subconditions may be used for the construction. For conjunctive constraints we take repair programs for the components and compose them to a typed repair program, provided that there is a sequentialization that preserves the preceding constraints. For disjunctive constraints we need a repair program for one component. Altogether, for so-called legit constraints, we can construct a repair program. These constructions are done in the \mathcal{M} -adhesive category of typed graphs with \mathcal{E}' - \mathcal{M} -pair factorization.

A model graph based on the Eclipse Modeling Framework (EMF) [24], short EMF model graph, is a typed graph satisfying some structural EMF constraints. Application of the results for typed graphs to the EMF world yields model completion programs for EMFk constraints, a first-order version of the EMF constraints, such that the application to a typed graph yields an EMFk model graph. The results known from typed graph repair are applied to EMFk model repair and EMF model repair.

The structure of the paper is as follows. In Section 2, we review the definitions of typed graphs, typed graph conditions, and typed graph programs. In Section 3, we introduce the concept of typed repair programs and show that there are repair programs for a large number of conditions, so-called legit conditions. Application of a typed repair program to any typed graph yields a typed graph satisfying the constraint. In Section 4, application of the results to EMF-world yields model-repair and completion programs for EMFk constraints, a first-order variant of EMF constraints. From these results, we derive results of EMF-model repair and completion. In Section 5, we present some related concepts. In Section 6, we give a conclusion and mention some further work.

2 Preliminaries

In the following, we recall the definitions of typed graphs, graph conditions, rules and transformations, graph programs, and basic transformations [2, 6, 17]. In the following, our concepts are based on [2]. For simplicity, we ignore the attributes.

A directed graph consists of a set of nodes and a set of edges where each edge is equipped with a source and a target node.

Definition 1 (graphs & morphisms). A (*directed*) graph $G = (V_G, E_G, s_G, t_G)$ consists of a set V_G of nodes and a set E_G of edges, as well as source and target functions $s_G, t_G: E_G \rightarrow V_G$. Given graphs G and H , a (*graph*) morphism $g: G \rightarrow H$ consists of total functions $g_V: V_G \rightarrow V_H$ and $g_E: E_G \rightarrow E_H$ that preserve sources and targets, that is, $g_V \circ s_G = s_H \circ g_E$ and $g_V \circ t_G = t_H \circ g_E$. The morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is injective and surjective. In the latter case, G and H are *isomorphic*, denoted by $G \cong H$.

Convention. Drawing a graph, nodes are drawn as circles and edges as arrows. Arbitrary morphisms are drawn by usual arrows \rightarrow , injective ones by \hookrightarrow .

A type graph (with containment) is a graph with a distinguished set of containment edges, and a relation of opposite edges.

Definition 2 (Type graph). A type graph $TG = (T, C, O)$ consists of a graph T , a set $C \subseteq E_T$ of *containment edges*, and a relation $O \subseteq E_T \times E_T$ of *opposite edges*. The relation O is anti-reflexive, symmetric, functional, i.e., $\forall (e_1, e_2), (e_1, e_3) \in O, e_2 = e_3$, and opposite direction, i.e., $\forall (e_1, e_2) \in O, s(e_1) = t(e_2)$ and $s(e_2) = t(e_1)$.

Convention. The drawing of a type graph is obtained from the underlying graph by marking every containment edge (\hookrightarrow) with a black diamond at the source, and adding, for every pair (e_1, e_2) of opposite edges, a bidirectional edge (\longleftrightarrow) between the source and the target of the first edge with two edge type names, one at each end.

Example 1. A type graph for Petri-nets is given in Figure 2.

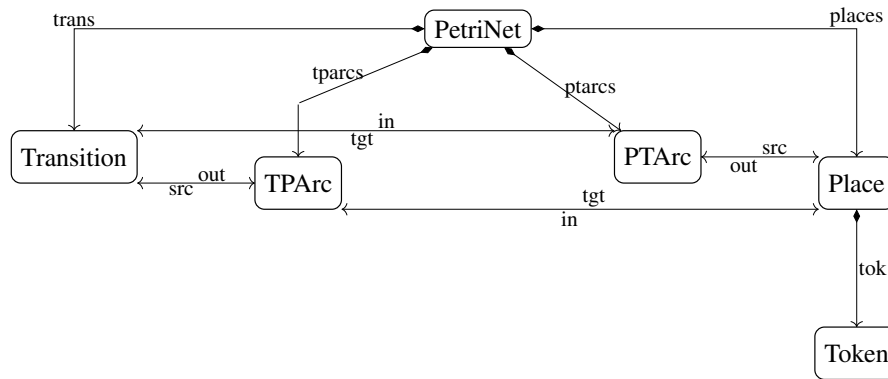


Figure 2: Type graph for Petri-nets, adapted from [27]

The type graph consists of the nodes PetriNet (PN), Place (Pl), Transition (Tr), Token (Tk), place-to-transition arcs (PTArc), and transition-to-place arcs (TPArc), written inside the nodes, and the edges places, trans, and tok. The distinguished containment edges from the PetriNet to the Place (Transition, PTArc, and TPArc)-node are marked in the graph. The opposite edge relation relates the edges from the PTArc (TPArc)-node to the Place (Transition)-node of type src and the Place (Transition)-node to the PTArc (TPArc)-node of type out.

Assumption. In the following, let $TG = (T, C, O)$ be a fixed type graph.

A typed graph over a type graph is a graph together with a typing morphism. The typing itself is done by a graph morphism between the graph and the type graph.

Definition 3 (typed graphs). A *typed graph* $(G, type)$ is a graph G together with a typing morphism $type: G \rightarrow T$. Given typed graphs $(G, type_G)$, $(H, type_H)$, a *typed graph morphism* $g: G \rightarrow H$ is a graph morphism such that $type_{H,V} \circ g_V = type_{G,V}$ and $type_{H,E} \circ g_E = type_{G,E}$. For a node v (an edge e) in G , $type_V(v)$ ($type_E(e)$) is the *node (edge) type*.

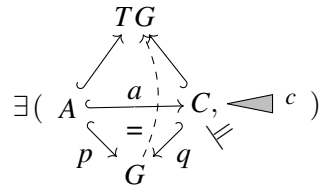
Convention. Given a typed graph $(G, type)$, we draw the graph G and put in type information: For a node v in G , we depict the node type $type_V(v)$ inside the node; for an edge e in G , we depict the edge type $type_E(e)$ near the target node of the edge e . Each edge with edge type containment edge is marked as a containment edge. For every pair of nodes whose type nodes are connected by an opposite edge, an opposite edge is added. For each pair (v_1, v_2) of nodes in G for which $type_V(v_1), type_V(v_2)$ are connected by an opposite edge, a bidirectional edge between the nodes with two edge type names, one at each end, is added.

Assumption. In the following, all graphs are typed over TG and all morphisms are injective.

Note. Typed graphs (over TG) with containment and morphisms form a category \mathbf{Graphs}_{TG} . This is \mathcal{M} -adhesive and has a \mathcal{E}' - \mathcal{M} pair factorization [5, 3] where \mathcal{M} is the class of injective morphisms and \mathcal{E}' is the class of pairs of jointly surjective morphisms. \mathcal{M} -adhesiveness implies the existence of pushouts (used in Definition 5 and Lemma 1); \mathcal{E}' - \mathcal{M} pair factorization is used in the shift construction in Lemma 1.

Typed graph conditions are nested constructs, which can be represented as trees of morphisms equipped with quantifiers and Boolean connectives. Graph conditions and first-order graph formulas are expressively equivalent.

Definition 4 (typed graph conditions). A (typed graph) condition over a graph A is of the form (a) true or $\exists(a, c)$ where $a: A \hookrightarrow C$ is a real inclusion morphism, i.e., $A \subset C$, and c is a condition over C . (b) For a condition c over A , $\neg c$ is a condition over A . (c) For conditions c_i ($i \in I$ for some finite index set I) over A , $\bigwedge_{i \in I} c_i$ is a condition over A . Conditions over the empty graph \emptyset are called *constraints*. In the context of rules, conditions are called *application conditions*. Any morphism $p: A \hookrightarrow G$ satisfies true . A morphism p satisfies $\exists(a, c)$ with $a: A \hookrightarrow C$ if there exists an morphism $q: C \hookrightarrow G$ such that $q \circ a = p$ and q satisfies c .



A morphism p satisfies $\neg c$ if p does not satisfy c , and p satisfies $\bigwedge_{i \in I} c_i$ if p satisfies each c_i ($i \in I$). We write $p \models c$ if p satisfies the condition c (over A). A graph G satisfies a constraint c , $G \models c$, if the morphism $p: \emptyset \hookrightarrow G$ satisfies c . A constraint c is *satisfiable* if there is a graph G that satisfies c .

Notation. Conditions may be written in a more compact form: $\exists a := \exists(a, \text{true})$, $\text{false} := \neg \text{true}$, $\forall(a, c) := \nexists(a, \neg c)$, and $\nexists := \neg \exists$. The expressions $\bigvee_{i \in I} c_i$ and $c \Rightarrow c'$ are defined as usual. For a morphism $a: A \hookrightarrow C$ in a condition, we just depict the codomain C , if the domain A can be unambiguously inferred.

The following is done in the framework of \mathcal{M} -adhesive categories. Rules are specified by a pair of morphisms, interface morphisms, and an application condition. By the interfaces, it becomes possible to hand over information between the transformation steps. In contrast to [25], our vertical morphisms are injective.

Definition 5 (typed rules & transformations). Given a category \mathcal{C} , a (typed) rule $\rho = \langle x, p, ac, y \rangle$ (with interfaces X, Y) consists of a plain rule $p = \langle L \hookrightarrow K \hookrightarrow R \rangle$ of morphisms $l: K \hookrightarrow L, r: K \hookrightarrow R$, morphisms

$x: X \hookrightarrow L, y: Y \hookrightarrow R$, the (*left and right*) *interface morphisms*, and a left application condition ac over L . The partial morphism $i: X \hookrightarrow Y$ with $i = y^{-1} \circ r \circ l^{-1} \circ x$ is the *interface morphism* of the rule ρ .

If the domain of an interface morphisms is empty or the application condition ac is true , then the component may not be written.

A *direct transformation* from G to H applying ρ at $g: X \hookrightarrow G$ consists of the following steps:

- (1) Mark a morphism $g': L \hookrightarrow G$, called *match* satisfying the dangling condition, such that $g = g' \circ x$ and $g' \models ac$.
- (2) Apply the plain rule p at g' yielding a morphism $h': R \hookrightarrow H$.
- (3) Unmark $h: Y \hookrightarrow H$, i.e., define $h = h' \circ y$.

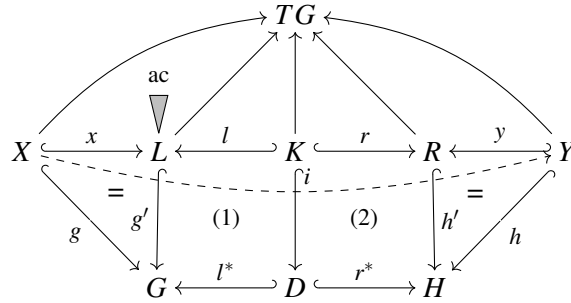


Figure 3: A direct transformation

The application of a plain rule is as in the double-pushout approach [3] in the category of typed graphs. A plain rule $p = \langle L \hookrightarrow K \hookrightarrow R \rangle$ is applicable to a graph G w.r.t. a morphism $g': L \hookrightarrow G$, iff g' satisfies the *dangling condition*: “No edge in $G - g'(L)$ is incident to a node in $g'(L - K)$.”

The *semantics* of the rule ρ is the set $\llbracket \rho \rrbracket$ of all triples $\langle g, h, i \rangle$ of a morphism $g: X \hookrightarrow G$, a morphism $h: Y \hookrightarrow H$, and a partial interface morphism $i: X \hookrightarrow Y$ with $i = y^{-1} \circ r \circ l^{-1} \circ x$. Instead of $\langle g, h, i \rangle \in \llbracket \rho \rrbracket$, we write $g \Rightarrow_{\rho, i} h$ or short $g \Rightarrow_{\rho} h$.

The dangling edges operator. For node-deleting rules ρ , the dangling condition may not be satisfied. In this case, ρ' means that the rule shall be applied in the SPO-style of replacement [10], i.e., first to remove the dangling edges, and, afterwards application of the rule in the DPO style. Note that this style of replacement also can be described by a DPO-program that fixes a match for the rule, deletes the dangling edges, and afterwards applies the rule at the match. The proceeding can be extended to sets of rules: For a rule set \mathcal{S} , $\mathcal{S}' = \{\rho' \mid \rho \in \mathcal{S}\}$.

Interface & markings. Rules with interfaces enable the control over marking and unmarking of elements in a typed graph and are capable of handling the markings over transformation steps. The left interface restricts the application of the rule to a previously marked context: Given a morphism $g: X \hookrightarrow G$, the application is restricted to those morphisms $g': L \hookrightarrow G$ that fit to g , i.e. $g = g' \circ x$. The right interface restricts the application of the next rule: By the morphism $h: Y \hookrightarrow H$, the next rule can only be applied at Y . Instead of rules with interfaces in the sense of [17], we could use markings as, e.g., in [7, 18]. We have decided to use the interfaces instead of markings, because we have nested markings and the description of markings by morphisms makes transparent what happens.

Typed graph programs are made of sets of typed rules with interface, non-deterministic choice $\{P, Q\}$, sequential composition $\langle P; Q \rangle$, the try-statement $\text{try } P$, and the as long as possible iteration $P \downarrow$.

Definition 6 (typed graph programs). The set of (typed graph) programs with interface X , $\text{Prog}(X)$, is defined inductively:

- (1) Every typed rule ρ with interface X (and Y) is in $\text{Prog}(X)$.
- (2) If $P, Q \in \text{Prog}(X)$, then $\{P, Q\}$ is in $\text{Prog}(X)$.
- (3) If $P \in \text{Prog}(X)$ and $Q \in \text{Prog}(Y)$, then $\langle P; Q \rangle \in \text{Prog}(X)$.
- (4) If $P \in \text{Prog}(X)$, then $\text{try } P$, and $P \downarrow$ are in $\text{Prog}(X)$.

The statement Skip denotes the identity rule $\text{id}_X = \langle X \leftarrow X \rightarrow X \rangle$.

The *semantics* of a program P with interface X , denoted by $\llbracket P \rrbracket$, is a set of triples such that, for all $\langle g, h, i \rangle \in \llbracket P \rrbracket$, the domain of g and i is X and the codomain of h and i is equal:

- (1) $\llbracket \rho \rrbracket$ as in Definition 5,
- (2) $\llbracket \{P, Q\} \rrbracket = \llbracket P \rrbracket \cup \llbracket Q \rrbracket$,
- (3) $\llbracket \langle P; Q \rangle \rrbracket = \{ \langle g_1, h_2, i_2 \circ i_1 \rangle \mid \langle g_1, h_1, i_1 \rangle \in \llbracket P \rrbracket, \langle g_2, h_2, i_2 \rangle \in \llbracket Q \rrbracket, h_1 = g_2 \}$,
- (4) $\llbracket \text{try } P \rrbracket = \{ \langle g, h, i \rangle \mid \langle g, h, i \rangle \in \llbracket P \rrbracket \} \cup \{ \langle g, g, \text{id} \rangle \mid \nexists h. \langle g, h, i \rangle \in \llbracket P \rrbracket \}$,
 $\llbracket P \downarrow \rrbracket = \{ \langle g, h, \text{id} \rangle \in P^* \mid \nexists h'. \langle h, h', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket \}$,

where $P^* = \bigcup_{j=0}^{\infty} P^j$ with $P^0 = \text{Skip}$, $P^j = \langle \text{Fix}(P); P^{j-1} \rangle$ for $j > 0$ and $\llbracket \text{Fix}(P) \rrbracket = \{ \langle g, h \circ i, \text{id} \rangle \mid \langle g, h, i \rangle \in \llbracket P \rrbracket \}$. Instead of $\langle g, h, i \rangle \in \llbracket P \rrbracket$, we write $g \Rightarrow_{P, i} h$ or short $g \Rightarrow_P h$.

In the following, we consider the basic transformations [6]. The construction Shift “shifts” existential conditions over morphisms into a disjunction of existential application conditions. This can be done because the category of typed graphs has an $\mathcal{E}^l - \mathcal{M}$ -factorization. The construction Left “shifts” a right application condition over a rule into a left application condition. Constraints can be integrated into left application conditions of a rule such that every transformation is condition-preserving.

Lemma 1 (Shift, Left, Pres). In an \mathcal{M} -adhesive category \mathcal{C} with $\mathcal{E}^l - \mathcal{M}$ pair factorization, there are constructions Shift , Left , and Pres such that the following holds. For each condition d over P and every morphism $b: P \hookrightarrow R, n: R \hookrightarrow H, n \circ b \models d \iff n \models \text{Shift}(b, d)$. For each rule $p = \langle L \leftarrow K \rightarrow R \rangle$ and each condition ac over R , for each $G \Rightarrow_{p, g, h} H, g \models \text{Left}(p, \text{ac}) \iff h \models \text{ac}$. For each rule ρ and each condition d , a condition $\text{ac} = \text{Pres}(\rho, d)$ can be constructed such that the rule $\langle \rho, \text{ac} \rangle$ is d -preserving, i.e., for all $g \Rightarrow_{\langle \rho, \text{ac} \rangle} h, g \models d$ implies $h \models d$.

A pair (a', b') of morphisms is *jointly surjective* if for each $x \in R'$ there is a preimage $y \in R$ with $a'(y) = x$ or $z \in C$ with $b'(z) = x$. For a rule $p = \langle L \leftarrow K \rightarrow R \rangle$, $p^{-1} = \langle R \leftarrow K \rightarrow L \rangle$ denotes the *inverse* rule. For $L' \Rightarrow_p R'$ with intermediate graph K' , $\langle L' \leftarrow K' \rightarrow R' \rangle$ is the *derived* rule.

Construction 1. The construction is as follows.

$$\begin{array}{ccc}
 P \xrightarrow{b} R & & \text{Shift}(b, \text{true}) := \text{true}. \\
 \downarrow a & \text{(0)} & \downarrow a' \\
 C \xrightarrow{b'} R' & & \text{Shift}(b, \exists(a, c)) := \bigvee_{(a', b') \in \mathcal{F}} \exists(a', \text{Shift}(b', c)) \text{ where} \\
 \triangle_c & & \mathcal{F} = \{ (a', b') \mid b' \circ a = a' \circ b, a', b' \text{ inj}, (a', b') \text{ jointly surjective} \} \\
 & & \text{Shift}(b, \neg d) := \neg \text{Shift}(b, d), \text{Shift}(b, \wedge_{i \in I} d_i) := \wedge_{i \in I} \text{Shift}(b, d_i).
 \end{array}$$

$$\begin{array}{ccc}
R \longleftarrow K \longrightarrow L & & \text{Left}(p, \text{true}) := \text{true}. \\
a \downarrow (1) \quad \downarrow (2) \quad \downarrow d' & & \text{Left}(p, \exists(a, \text{ac})) := \exists(a', \text{Left}(p', \text{ac})) \text{ if } p^{-1} \text{ is applicable w.r.t. the mor-} \\
R' \longleftarrow K' \longrightarrow L' & & \text{phism } a, p' := \langle L' \longleftarrow K' \longrightarrow R' \rangle \text{ is the } \textit{derived} \text{ rule, and } \text{false}, \text{ otherwise.} \\
\triangle_{\text{ac}} \quad \quad \quad \triangle & & \text{Left}(p, \neg \text{ac}) := \neg \text{Left}(p, \text{ac}). \quad \text{Left}(p, \wedge_{i \in I} \text{ac}_i) := \wedge_{i \in I} \text{Left}(p, \text{ac}_i).
\end{array}$$

$$\text{Pres}(\rho, d) := \text{Shift}(A \hookrightarrow L, d) \Rightarrow \text{Left}(\rho, \text{Shift}(A \hookrightarrow R, d)).$$

Example 2. Let $\rho = \langle \boxed{\text{Pl}} \boxed{\text{Tk}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \rangle$ be a rule, $d = \#(\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xrightarrow{\text{tok}} \boxed{\text{Pl}})$ be a constraint, and b_L and b_R be the morphisms from the empty graph to the left- and right-hand side of the rule, respectively. Then we have the following.

$$\begin{aligned}
\text{Shift}(b_L, d) &= \#(\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xrightarrow{\text{tok}} \boxed{\text{Pl}}) \wedge \dots \\
\text{Shift}(b_R, d) &= \#(\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xrightarrow{\text{tok}} \boxed{\text{Pl}}) \wedge \dots \\
\text{Left}(\rho, \text{Shift}(b_R, d)) &= \#(\boxed{\text{Pl}} \quad \boxed{\text{Tk}} \xrightarrow{\text{tok}} \boxed{\text{Pl}}) \wedge \dots \\
\text{Pres}(\rho, d) &= \#(\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xrightarrow{\text{tok}} \boxed{\text{Pl}}) \wedge \dots \Rightarrow \#(\boxed{\text{Pl}} \quad \boxed{\text{Tk}} \xrightarrow{\text{tok}} \boxed{\text{Pl}}) \wedge \dots
\end{aligned}$$

If the rule ρ is equipped with the application condition $\text{Pres}(\rho, d)$, we obtain the rule $\rho' = \langle \rho, \text{Pres}(\rho, d) \rangle$, restricting the applicability of the rule to those matches satisfying the application condition and preserving the constraint d . The application condition is satisfied if the rule is applied to an occurrence of a graph with Tk-node that does not have incoming containment edges from different Pl-nodes. (A node of type Tk is said to be *Tk-node*.)

3 Repair programs

In this section, we introduce repair programs and show some repair results for repair programs.

A repair program for a constraint is a graph program with the property that there exists a derivation and the application to any graph yields a graph satisfying the constraint. More generally, we consider repair programs for conditions.

Definition 7 (repair programs). A (typed) program P is a (typed) *repair program* for a constraint d if, for all (typed) graphs G , $\exists G \Rightarrow_P H$ and $\forall G \Rightarrow_P H$, $H \models d$. A program P with interface A is a *repair program* for a condition ac over A , if, for all injective morphisms $g: A \hookrightarrow G$, $\exists g \Rightarrow_{P_i} h$ and $\forall g \Rightarrow_{P_i} h$, $h \circ i \models \text{ac}$.

Example 3. For the constraint d (see below), intuitively meaning, there do not exist two parallel edges of type tok between a Pl-node and a Tk-node, the program P_d is a repair program for d .

$$d = \#(\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}) \quad P_d = \langle \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \rangle \downarrow$$

It works as follows: whenever there are two parallel tok-edges, P_d deletes one of the two tok-edges as long as possible.

We look for stable, maximally preserving, and terminating repair programs. A repair program is *stable*, if it does nothing whenever the condition is already satisfied, *maximally preserving*, if, informally, items are preserved whenever possible (see [21]), *terminating* if the relation \Rightarrow is terminating.

We start with basic conditions of the form $\exists a$ (or $\nexists a$) with morphism $a: A \hookrightarrow C$. For basic conditions, we construct so-called *repairing* sets from the morphism of the condition and repair programs based on the repairing set using the `try`-statement and the as-long-as-possible iteration, respectively.

Lemma 2 (basic repair). For basic conditions, there are repair programs.

Construction 2. For a real morphism $a: A \hookrightarrow C$, the programs $P_{\exists a}$ and $P_{\nexists a}$ are constructed as follows.

$$\begin{aligned} P_{\exists a} &= \text{try } \mathcal{R}_a \quad \text{with } \mathcal{R}_a = \{ \langle b, B \Rightarrow C, \text{ac} \wedge \text{ac}_B, a \rangle \mid A \hookrightarrow^b B \subset C \} \\ P_{\nexists a} &= \mathcal{S}'_a \downarrow \quad \text{with } \mathcal{S}_a = \{ \langle a, C \Rightarrow B, b \rangle \mid A \hookrightarrow^b B \subset C \text{ and } (*) \} \end{aligned}$$

where $\text{ac} = \text{Shift}(A \hookrightarrow B, \nexists a)$, $\text{ac}_B = \bigwedge_{B \subset B' \subset C} \nexists B'$, $(*)$ if $E_C \supset E_A$ then $|V_C| = |V_B|$, $|E_C| = |E_B| + 1$ else $|V_C| = |V_B| + 1$, and $'$ denotes the dangling edges operator.

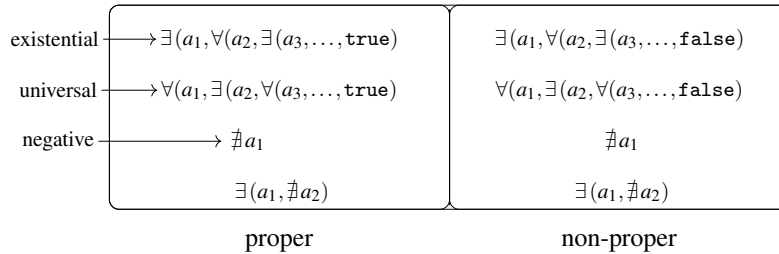
The rules in \mathcal{R}_a are of the form $B \Rightarrow C$ where $A \subseteq B \subset C$. They possess an application condition ac requiring the condition $\nexists a$, shifted from A to B , and the application condition ac_B requiring that no larger subgraph B' of C occurs. The rules in \mathcal{S}_a are of the form $C \Rightarrow B$ where $A \subseteq B \subset C$ such that, if the number of edges in C is larger than the one in A , they delete one edge and no node, and delete a node, otherwise. By $B \subset C$, both rule sets do not contain identical rules.

Example 4. Consider the condition $d = \exists b$ with $b: \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$, intuitively meaning that, whenever there is a place there exists a token and a connecting containment edge. Application of the Construction 2 yields a rule set \mathcal{R}_b with two rules.

$$\mathcal{R}_b = \left\{ \begin{array}{l} \rho_1 = \langle x_1, \boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}, \nexists \boxed{\text{Pl}} \boxed{\text{Tk}}, y_1 \rangle \\ \rho_2 = \langle x_2, \boxed{\text{Pl}} \boxed{\text{Tk}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}, \nexists \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \wedge \nexists \boxed{\text{Tk}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}, y_2 \rangle \end{array} \right.$$

where $x_1: \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}$, $y_1: \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$, $x_2: \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \boxed{\text{Tk}}$, and $y_2: \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$. The rule ρ_1 requires a node of type Pl and attaches a node of type Tk and a connecting containment edge, provided that there do not exist a Pl-node and a Tk-node. The second rule ρ_2 requires an occurrence of a Pl- and a Tk-node and inserts a connecting containment edge, provided there is no containment edge from the occurrence of the Pl-node to the image of Tk-node, and there is no containment edge to another Tk-node. By Lemma 2 resp. Theorem 1(2), $P_d = \text{try } \mathcal{R}_b$ is a repair program for $d = \exists b$.

Conditions with alternating quantifiers ending with `true` or of the form $\exists(a, \nexists b)$ or $\nexists b$ are *proper*. A proper condition of the form $\forall(a, c)$ and $\exists(a, c)$ that ends with `true` is *universal* and *existential*, respectively. A condition of the form $\exists a$ ($\nexists a$) is *positive* (*negative*).



For proper conditions, a repair program can be constructed.

Theorem 1 (Repair I). There is a repair program for proper conditions.

Construction 3. For proper conditions d , the repair program P_d is constructed inductively as follows.

- (1) For $d = \text{true}$, $P_d = \text{Skip}$.
- (2) For $d = \exists a$, $P_d = \text{try } \mathcal{R}_a$.
- (3) For $d = \nexists a$, $P_d = \mathcal{S}'_a \downarrow$.
- (4) For $d = \exists(a, c)$, $P_d = P_{\exists a}; \langle \text{Mark}(a); P_c; \text{Unmark}(a) \rangle$.
- (5) For $d = \forall(a, c)$, $P_d = \langle \text{Mark}(a, \neg c); P_c; \text{Unmark}(a) \rangle \downarrow$.

where $a: A \hookrightarrow C$ is real, \mathcal{R}_a and \mathcal{S}'_a are the sets according to Construction 2, and P_c is a repair program for c with interfaces C . $\text{Mark}(a) = \langle a, \text{id}_C \rangle$ is the rule with left interface a and identical plain rule $\text{id}_C = \langle C \hookrightarrow C \hookrightarrow C \rangle$. Given an occurrence of A , it is used for a marking of an occurrence of C , extending the occurrence of A . Similar, $\text{Mark}(a, ac) = \langle a, \text{id}_C, ac \rangle$ is used for marking an occurrence of C satisfying the condition ac . $\text{Unmark}(a) = \langle \text{id}_C, a \rangle$ is the identical plain rule with right interface a , used for unmarking the occurrence of C .

Example 5. Given the constraint $d = \forall(\boxed{\text{Pl}} , \exists(\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}))$, meaning that, for each place, there exists a token, a repair program for d can be constructed according to Theorem 1. The constraint d is of the form $\forall(a, c)$ with morphism $a: \emptyset \hookrightarrow \boxed{\text{Pl}}$ and condition $c = \exists \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$. By Theorem 1(5), $P_d = \langle \text{Mark}(a, \neg c); P_c; \text{Unmark}(a) \rangle \downarrow$. The condition c is of the form $\exists b$. By Repair Theorem 1(2), the repair program for c is $P_c = \text{try } \mathcal{R}_b$, where \mathcal{R}_b is the rule set from Example 4. The program P_d marks an occurrence of a Pl-node without connecting containment edge to a Tk-node. The program P_c tries to add a Tk-node and the containment edge, provided there do not exist a Pl-node and a Tk-node, and to add a containment edge between a Pl- and Tk-node, provided that there does not exist such an edge to another Tk-node. Finally the marked part is unmarked. This is done as long as possible. Whenever no further application is possible, the constraint d is satisfied.

The constructed programs are increasing (decreasing): A program P is *decreasing* (*increasing*) if all rules in P are decreasing (increasing). A rule $\rho = \langle L \hookrightarrow K \hookrightarrow R, ac \rangle$ is *decreasing* if $L \supset K \cong R$ and increasing if $L \cong K \subset R$.

Fact 1. For negative conditions, the repair program is decreasing. For positive, existential, and universal conditions, the repair program is increasing.

Proof (of Theorem 1). In [7, Theorem 1] the statement is proven for graphs. The statement also holds for typed graphs: For every morphism $a: A \hookrightarrow C$, and every proper subgraph B of C , define $\text{type}_B = \text{type}_C \circ \text{inc}_B$, where for $B \subseteq C$, inc_B (short i_B) denotes the inclusion of B in C . Then $\text{type}_A = \text{type}_B \circ \text{inc}_A$ and the rules $B \Rightarrow C \in \mathcal{R}_a$ and $C \Rightarrow B \in \mathcal{S}_a$ consist of typed graph morphisms. In this way the graphs in the rules in \mathcal{R}_a and \mathcal{S}_a become typed. The typing of the conditions is a direct consequence. \square

Remark. Theorem 1 could be formulated for a larger class of conditions. In Construction 3, it is not necessary that the condition c in a condition $\exists(a, c)$ (or $\forall(a, c)$) is proper. The construction of a repair program can be done provided that there exists a repair program for c . Properness only guarantees the existence of a repair program.

In the following, we consider conjunctions of conditions. We try to construct a repair program for a conjunction from the repair programs of the conditions in the conjunction.

Given a conjunction d of conditions, we proceed as follows.

- (1) Try to find a “preserving sequentialization” d_1, \dots, d_n of d .
- (2) Construct repair programs P_1, \dots, P_n for d_1, \dots, d_n .
- (3) Compose the repair programs to a repair program $P = \langle P_1; \dots; P_n \rangle$ for d .

1. For a conjunction of negative (positive) conditions, this is very simple. We take any sequentialization of the negative (positive) conditions and consider the sequential composition of the corresponding repair programs. This works because, for negative (positive) conditions, the repair programs are decreasing (increasing), and every sequence of decreasing (increasing) repair programs “preserves” the preceding negative (positive) conditions.

2. For conjunctions of universal conditions, this is not so easy: In general, not every sequentialization is preserving. Consider, e.g., the constraints $d_1 = \forall(\bullet, \exists \bullet \looparrowright)$ and $d_2 = \forall(\bullet, \exists \bullet \rightarrow \bullet)$ with the repair programs P_1 and P_2 constructed according to Construction 3. For the sequentialization d_1, d_2 , the program P_2 does not preserve the constraint d_1 : For a node with loop satisfying d_1 the program P_2 adds a new node and a connecting edge. The new node does not have a loop, i.e., the resulting graph does not satisfy d_1 . For the sequentialization d_2, d_1 , the program P_1 preserves the constraint d_2 .

3. Moreover, sometimes there is no preserving sequentialization. Consider, e.g., $d_1 = \forall(\bullet, \exists \bullet \rightarrow \bullet)$ and $d_2 = \forall(\bullet \rightarrow \bullet, \exists \bullet \rightarrow \bullet \bullet)$ with the repair programs P_1 and P_2 constructed according to Construction 3. The condition $d_1 \wedge d_2$ is satisfiable: the graph $\bullet \xrightarrow{\curvearrowright} \bullet \rightarrow \bullet$ satisfies $d_1 \wedge d_2$. Then P_2 does not preserve d_1 and P_1 does not preserve d_2 : Application of P_2 to $\bullet \xrightarrow{\curvearrowright} \bullet \rightarrow \bullet \models d_1$ yields to $\bullet \xrightarrow{\curvearrowright} \bullet \bullet \not\models d_1$ and application of P_1 to $\bullet \models d_2$ yields to $\bullet \xrightarrow{\curvearrowright} \bullet \not\models d_2$.

For this proceeding, preservation of conditions is essential: Whenever a condition is satisfied, it shall be preserved in the following.

Definition 8 (preservation). A program P is d -preserving if every rule in P is d -preserving.

Lemma 3 (preservation). For every program P and every d , there is a d -preserving program P^d .

Construction 4. P^d : replace all rules ρ in P by $\langle \rho, \text{Pres}(\rho, d) \rangle$ (Construction 1).

Proof. By Construction 1, all rules in P^d are d -preserving, thus, the program is d -preserving. \square

In the following, we consider a sequence of conditions together with their repair programs.

Convention. Let $ds = d_1, \dots, d_n$, $Ps = P_1, \dots, P_n$, and P_i be a repair program for d_i , respectively.

A sequence of programs is preserving if for each natural number k , the respective repair program P_k preserves all preceding conditions, i.e. P_1 is a repair program for d_1 , P_2 is a repair program for d_2 and d_1 -preserving, P_3 is a repair program for d_3 and $d_1 \wedge d_2$ -preserving, and so on.

Definition 9 (preservation). The sequence Ps is ds -preserving (and the sequence ds is preserving) if, for $k = 2, \dots, n$, P_k is $\bigwedge_{i=1}^{k-1} d_i$ -preserving.

We show that sequences of repair programs for sequences of conditions can be sequentially composed to a repair program for the conjunction, provided that the sequences of conditions is preserving.

Lemma 4 (preserving repair). If d_1, \dots, d_n -preserving, then $\langle P_1; \dots; P_n \rangle$ is a repair program for $\bigwedge_{i=1}^n d_i$.

Proof. By induction on the number n of conditions with the repair programs. For $n=1$, by Theorem 1, P_1 is a repair program for d_1 . Inductive hypothesis: If P_2, \dots, P_n is d_1, \dots, d_n -preserving, then $P = \langle P_1; \dots; P_n \rangle$ is a repair program for the conjunction $\bigwedge_{i=1}^n d_i$. Inductive step: For $n = n + 1$, let P_2, \dots, P_{n+1} be d_1, \dots, d_{n+1} -preserving. Then P_2, \dots, P_n is d_1, \dots, d_n -preserving and, by induction hypothesis, the program $P = \langle P_1; \dots; P_n \rangle$ is a repair program for the conjunction $d = \bigwedge_{i=1}^n d_i$. Moreover, P_{n+1} is a d -preserving repair program for d_{n+1} . Consequently, for every transformation $g \Rightarrow_P g_n \Rightarrow_{P_{n+1}} h$, $g_n \models d$ and $h \models \bigwedge_{i=1}^{n+1} d_i$. Thus, $\langle P_1; \dots; P_{n+1} \rangle$ is a repair program for $\bigwedge_{i=1}^{n+1} d_i$. \square

A sequence ds of conditions is *negative (or positive, or universal)* if all conditions in it have the property. For a sequence of negative (or positive) conditions, the sequence Ps of repair programs is ds -preserving.

Fact 2. If ds is negative (or positive), then Ps is ds -preserving.

In the following, we consider a conjunction of negative and universal conditions. Let e_1 be the conjunction of negative and e_2 the conjunction of universal conditions. By Fact 2, every sequence ds_1 of negative conditions is preserving and, by Lemma 4, the sequential composition $Q_1 = \langle P_1; \dots; P_k \rangle$ of the repair programs forms a repair program for e_1 . In general, not every sequentialization ds_2 of universal conditions is preserving. We have to require preservation. In the case of preservation, the sequential composition $Q_2 = \langle P_{k+1}; \dots; P_n \rangle$ of the repair programs forms a repair program for e_2 . But the repair program Q_2 may be not e_1 -preserving. By Lemma 5 below, Q_2 can be modified to an e_1 -preserving repair program $Q_2^{e_1}$ for e_2 . The idea is to delete all occurrences of the morphism a of the universal condition $\forall(a, c)$, which violate the condition c . Given a universal condition, we mark an occurrence of the morphism violating the condition, then the occurrence of the morphism at that position is deleted. To mark the morphism at that position, we modify the left interface to the identity of the codomain of the morphism. For a conjunction of universal conditions, an e_1 -preserving repair program can be constructed from the e_1 -preserving program by destroying all non-repaired occurrences of all the universal conditions. By Lemma 4, the program $\langle Q_1, Q_2^{e_1} \rangle$ becomes a repair program for $e_1 \wedge e_2$.

A conjunction is *negative (universal)* if all conditions in the conjunction are negative (universal).

Lemma 5 (preserving repair program). If Q_2 is a repair program for a preserving universal conjunction e_2 and e_1 is a negative conjunction, then there is an e_1 -preserving repair program $Q_2^{e_1}$ for e_2 .

Construction 5. For a repair program P for $\forall(a, c)$, let $P^{e_1} = \langle P^{e_1}; P_{\#a}^{\text{id}} \rangle$ where $P_{\#a}^{\text{id}} = \langle \text{Mark}(a, \neg c); \mathcal{S}_a^{\text{id}} \rangle \downarrow$ and $\mathcal{S}_a^{\text{id}}$ is obtained from \mathcal{S}_a by replacing the left interface morphism $a: A \hookrightarrow C$ by the identity $\text{id}: C \hookrightarrow C$. For a preserving conjunction e_2 of universal conditions with sequentialization d_1, \dots, d_n and repair program $Q_2 = \langle P_1; \dots; P_n \rangle$, let $Q_2^{e_1} = \langle P_1^{e_1}; \dots; P_n^{e_1} \rangle$.

Proof. 1. For a universal condition $\forall(a, c)$ with $a: A \hookrightarrow C$, the programs P and P^{e_1} are increasing. By the e_1 -preserving application condition, whenever the increasing program P^{e_1} is not a repair program, the condition c is not satisfied, and the decreasing program $P_{\#a}^{\text{id}}$ becomes applicable and destroys all occurrences that do not satisfy the condition c . Since e_1 is a conjunction of negative conditions, $P_{\#a}^{\text{id}}$ is d_1 -preserving. Consequently, P^{e_1} is e_1 -preserving. Then P^{e_1} is a repair program for d : For every occurrence of a , the occurrence is either (1) repaired by P^{e_1} or (2) destroyed by $P_{\#a}^{\text{id}}$.

2. By assumption, d_1, \dots, d_n is preserving. Moreover, for $i = 1, \dots, n$, P_i is repair program for d_i and, by Lemma 5.1, $P_i^{e_1}$ is an e_1 -preserving repair program for d_i . Since d_1, \dots, d_n is preserving and by Lemma 4, $Q_2^{e_1} = \langle P_1^{e_1}; \dots; P_n^{e_1} \rangle$ is a repair program for $\bigwedge_{i=1}^n d_i = e_2$. Since all programs in the sequential composition are e_1 -preserving, the program $Q_2^{e_1}$ is e_1 -preserving. Consequently, every transformation $g \Rightarrow_Q m$ is of the form $g \Rightarrow_{Q_1} h \Rightarrow_{Q_2^{e_1}} m$. Since Q_1 is a repair program for e_1 , $h \models e_1$. Since $Q_2^{e_1}$ is the e_1 -preserving repair program for e_2 , $m \models e_1 \wedge e_2$. Thus, Q is a repair program for $e_1 \wedge e_2$. \square

Fact 3 (composition). For arbitrary conditions e_1, e_2 , the following holds. If Q_1 is a repair program for e_1 and $Q_2^{e_1}$ be an e_1 -preserving repair program for e_2 , then $\langle Q_1, Q_2^{e_1} \rangle$ a repair program for $e_1 \wedge e_2$.

A sequence $ds = d_1, \dots, d_n$ is negative (or positive, or existential, or universal) if all d_i are negative (or positive, or existential, or universal). In the following, $ds_1 = d_1, \dots, d_k$, $ds_2 = d_{k+1}, \dots, d_n$ with conjunction $e_1 = \bigwedge_{i=1}^k d_i$ and $e_2 = \bigwedge_{i=k+1}^n d_i$.

The following theorem says under which conditions a repair program for a conjunction of conditions can be constructed from the repair programs of its components.

Theorem 2 (Repair II). There is a repair program P for a conjunction $d = \bigwedge_{i=1}^n d_i$ of conditions provided that d is satisfiable, there are repair programs P_1, \dots, P_n for d_1, \dots, d_n , respectively, and there is a sequentialization $ds = d_1, \dots, d_n$, and

1. ds is negative, or positive, or preserving,
2. ds_1 is positive, and ds_2 is existential (or universal) & preserving.
3. ds_1 is negative, and ds_2 is universal & preserving.

Construction 6.

1. For negative (or positive, or preserving) ds , let $P = \langle P_1; \dots; P_n \rangle$.
2. For positive ds_1 , universal (or existential) & preserving ds_2 , let $P = \langle Q_1; Q_2 \rangle$.
3. For negative ds_1 , universal & preserving ds_2 , let $P = \langle Q_1; Q_2^{e_1} \rangle$.

where P_1, \dots, P_n are repair programs for d_1, \dots, d_n , respectively, $Q_1 = \langle P_1; \dots; P_k \rangle$, $Q_2 = \langle P_{k+1}; \dots; P_n \rangle$, and $Q_2^{e_1} = \langle P_{k+1}^{e_1}; \dots; P_n^{e_1} \rangle$ where $e_1 = \bigwedge_{i=1}^k d_i$ and $e_2 = \bigwedge_{i=k+1}^n d_i$.

Example 6. Consider the constraints $d_1 = \nexists (\text{PI} \xrightarrow{\text{tok}} \text{TK} \xleftarrow{\text{tok}} \text{PI})$ and $d_2 = \forall (\text{PI} , \exists (\text{PI} \xrightarrow{\text{tok}} \text{TK}))$ (see Example 5). By Theorem 1, there are repair programs

$$\begin{aligned} P_1 &= \langle \text{PI} \xrightarrow{\text{tok}} \text{TK} \xleftarrow{\text{tok}} \text{PI} \Rightarrow \text{PI} \quad \text{TK} \xleftarrow{\text{tok}} \text{PI} \rangle \downarrow \\ P_2 &= \langle \text{Mark}(a, \neg c); \text{try } \mathcal{R}_b; \text{Unmark}(a) \rangle \downarrow \end{aligned}$$

where $a: \emptyset \hookrightarrow \text{PI}$, $c = \exists \text{PI} \hookrightarrow \text{PI} \xrightarrow{\text{tok}} \text{TK}$, and \mathcal{R}_b as in Example 4.

By Theorem 2, there is a repair program $P = \langle P_1; P_2^{d_1} \rangle$ for $d = d_1 \wedge d_2$. By Lemma 3, the d_1 -preserving version $P_2^{d_1}$ of P_2 is obtained from the repair program P_2 by equipping each rule ρ in \mathcal{R}_b with application condition $\text{Pres}(\rho, d_1)$. For $\rho_2 \in \mathcal{R}_b$, $\rho_2^{d_1} = \langle \rho_2, \text{Pres}(\rho_2, d_1) \rangle = \nexists (\text{PI} \xrightarrow{\text{tok}} \text{TK} \xleftarrow{\text{tok}} \text{PI} \wedge \dots \Rightarrow \nexists (\text{PI} \quad \text{TK} \xleftarrow{\text{tok}} \text{PI}) \wedge \dots$. The program $P_2^{d_1}$ is not a repair program for d_2 : $P_2^{d_1}$ is not applicable to the graph $G: \text{PI} \quad \text{TK} \xleftarrow{\text{tok}} \text{PI}$ and $G \not\models d_2$.

By Lemma 5, there is a d_1 -preserving repair program $P_2^{d_1} = \langle P_2^{d_1}; P_{\#a}^{\text{id}} \rangle$ for d_2 where $P_{\#a}^{\text{id}}$ is a slightly modified version of the repair program $P_{\#a}$ for the condition $\nexists a$. In more detail, the program looks

as follows: $P_{\#a}^{\text{id}} = \langle \text{Mark}(\boxed{\text{Pl}}, \# \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}); \langle x, \boxed{\text{Pl}} \Rightarrow \emptyset \rangle' \rangle \downarrow$ where x is the identity $x: \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}$. The program marks an occurrence of the Pl-node without incoming containment edge from a Tk-node and deletes (in SPO-style) the occurrence of the Pl-node; this is done as long as possible. In this way, Pl-nodes not satisfying the condition c , are deleted. We obtain a repair program for $d_1 \wedge d_2$.

Proof. 1. Let d_1, \dots, d_n be negative (positive). Then the repair programs P_1, \dots, P_n are decreasing (increasing) and d_1, \dots, d_n is preserving. Then, by Lemma 4, $\langle P_1; \dots; P_n \rangle$ is a repair program for $\wedge_{i=1}^n d_i$.

2. Let d_1, \dots, d_k be positive and d_{k+1}, \dots, d_n universal (or existential) and preserving. By Theorem 2.1 there are repair programs $Q_1 = \langle P_1; \dots; P_k \rangle$ and $Q_2 = \langle P_{k+1}; \dots; P_n \rangle$ for $e_1 = \wedge_{i=1}^k d_i$ and $e_2 = \wedge_{i=k+1}^n d_i$, respectively. Since Q_2 is increasing, it is e_1 -preserving. By Lemma 4, $\langle Q_1; Q_2 \rangle$ is a repair program for $e_1 \wedge e_2 = \wedge_{i=1}^n d_i$.

3. Let d_1, \dots, d_k be negative and d_{k+1}, \dots, d_n universal and preserving. By Theorem 2.1 there are repair programs $Q_1 = \langle P_1; \dots; P_k \rangle$ and $Q_2 = \langle P_{k+1}; \dots; P_n \rangle$ for $e_1 = \wedge_{i=1}^k d_i$ and $e_2 = \wedge_{i=k+1}^n d_i$, respectively. By Lemma 5, $Q_2^{e_1}$ is the e_1 -preserving repair program for e_2 . By Fact 3, $\langle Q_1; Q_2^{e_1} \rangle$ is a repair program for $e_1 \wedge e_2 = \wedge_{i=1}^n d_i$. An illustration of this part of the proof is given in Figure 4. \square

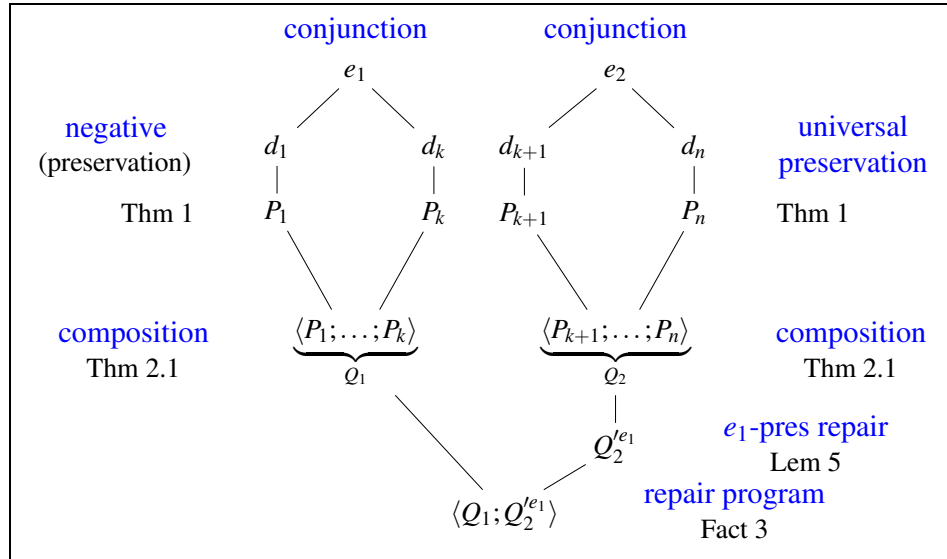


Figure 4: Illustration of the proof of Theorem 2.3

In the following, we consider *disjunctive* conditions, i.e. disjunctions of conditions. Whenever there exists a repair program for one of the subconditions, this repair program can be used for the disjunctive conditions as well. Every repair program for a condition is also a repair program for the corresponding disjunctive condition.

Theorem 3 (Repair III).

1. If P is a repair program for d and $d \Rightarrow d'$, then P is a repair program for d' .
2. Every repair program for d_1 is repair program for $\vee_{i=1}^n d_i$.
3. If P_1, \dots, P_n are repair programs for d_1, \dots, d_n , then $\{P_1, \dots, P_n\}$ is a repair program for $\vee_{i=1}^n d_i$.

Proof. 1. If P is a repair program for d and $d \Rightarrow d'$, then for every transformation $g \Rightarrow_P h$, $h \models d \Rightarrow d'$, i.e., P is a repair program for d' . 2. By $d_1 \Rightarrow \bigvee_{i=1}^n d_i$ and statement 1, P_1 is a repair program for $\bigvee_{i=1}^n d_i$. 3. Since $d_i \Rightarrow \bigvee_{i=1}^n d_i$ and by statement 1 and 2, $\{P_1, \dots, P_n\}$ is a repair program for $\bigvee_{i=1}^n d_i$ \square

There are repair programs for a large class of conditions: for all proper and generalized-proper ones, preserving conjunctions, and disjunctions. In this context, a condition is said to be generalized proper, if it is obtained from a proper one by replacing a subcondition by a condition (over the same graph) with a repair program.

Definition 10 (generalized proper). Let $d = Q(a, c)$ be a proper condition. A *generalized proper* condition $d' = [c/c']$ is obtained by replacing c with a condition c' , provided there exists a repair program for c' .

Definition 11 (legit conditions). The class of *legit* conditions is defined inductively as follows.

- (1) If d is proper or generalized proper, then d is legit.
- (2) If d_1, \dots, d_n are legit and preserving, then $\bigwedge_{i=1}^n d_i$ is legit.
- (3) If d_1 is legit, then $\bigvee_{i=1}^n d_i$ is legit.

Theorem 4 (Repair for legit conditions). For legit conditions, there is a repair program.

Proof. By induction of the structure of conditions. Let d be legit.

(1) If d is proper, then, by Theorem 1, there is a repair program for d .

If d is generalized proper, then d is of the form $Q(a, c)$ where c is legit. By induction, there is a repair program for c . By a generalized Theorem 1, there is a repair program for d . (2) Let $ds = d_1, \dots, d_n$ are legit and ds preserving. By induction hypothesis, there are repair programs for d_1, \dots, d_n . By Theorem 2, there is a repair program for the conjunction $\bigwedge_{i=1}^n d_i$. (3) Let d_1 be legit. By induction hypothesis, there is a repair program for d_1 . By Theorem 3, there is a repair program for $\bigvee_{i=1}^n d_i$. This completes the inductive proof. \square

As a consequence of Construction 3, we obtain the following.

Lemma 6 (program properties). The repair programs for proper conditions based on Construction 3 are (1) stable, (2) maximally preserving, (3) and terminating.

Proof. Let d be a proper condition and P_d the program in Construction 3. (1) By the application condition $ac = \text{Shift}(A \hookrightarrow B, \nabla a)$, a rule in \mathcal{R}_a can only be applied, iff the condition is not satisfied. By the semantics of `Skip`, `try`, and `↓`, the repair programs are stable. The proof of (2) and (3) can be found in [21] and [7], respectively. \square

Lemma 7 (program properties). The repair programs for legit conditions as above are stable and terminating.

Proof. By induction on the structure of conditions. Let d be a legit condition and P_d be the corresponding repair program.

(1) If d is proper, then by Lemma 6, the repair program P_d is stable and terminating. If d is generalized proper, then, by induction hypothesis, P_d is stable and terminating.

(2) By induction hypothesis, $P_i, P_i^{e_1}$ are stable and terminating, $\langle P_1; \dots; P_n \rangle$, and Q_1, Q_2 are stable and terminating. Finally $Q_2^{e_1} = \langle P_{k+1}^{e_1}; \dots; P_n^{e_1} \rangle$, with $P_j^{e_1} = \langle P_j^{e_1}; P_{\#a,j}^{\text{id}} \rangle$, is (a) stable and (b) terminating: (a) By Construction, $P_{\#a,j}^{\text{id}}$ is only applicable, iff the condition is not satisfied. By the semantics of \downarrow , it is stable. (b) By Construction, $P_{\#a,j}^{\text{id}}$ is decreasing, consequently it is terminating. Consequently, $\langle Q_1; Q_2^{e_1} \rangle$ is stable and terminating.

(3) If $d = \bigvee_i^n d_i$, then, by induction hypothesis, P_1 is stable and terminating. Consequently, $\{P_1, \dots, P_n\}$ is stable and terminating. \square

Remark (Implementation). The approach to graph repair has been implemented in ENFORCE+.

4 Application to meta-modeling

The standard tool for model-driven engineering is the Eclipse Modeling Framework (EMF). In [2], an EMF model graph is defined as a typed graph, representing the model, satisfying the following conditions: No node has more than one container. There are no two parallel edges of the same type. No cycles of containment occur. For all edges in the opposite edges relation, there exists an edge in opposite direction.

Definition 12 (EMF-model graph). A typed graph G is an *EMF-model graph*, if it satisfies the following conditions:

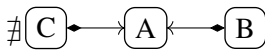
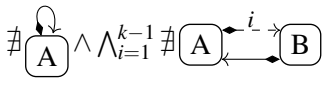
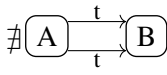
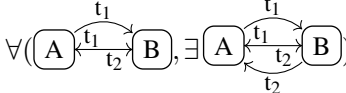
1. At most one container $\forall e_1, e_2 \in C_G. t_G(e_1) = t_G(e_2)$ implies $e_1 = e_2$
2. No containment cycle $\forall v \in V_G. (v, v) \notin \text{cont}_G$
3. No parallel edges $\forall e_1, e_2 \in E_G. s_G(e_1) = s_G(e_2), t_G(e_1) = t_G(e_2)$, and $\text{type}_{E_G}(e_1) = \text{type}_{E_G}(e_2)$ implies $e_1 = e_2$
4. All opposite edges $\forall (e_1, e_2) \in O. \forall e'_1 \in E_G. \text{type}_G(e'_1) = e_1. \exists e'_2 \in E_G. \text{type}_G(e'_2) = e_2, s_G(e'_1) = t_G(e'_2)$ and $s_G(e'_2) = t_G(e'_1)$

The set C_G denotes the set of edges in G which are typed by a containment edge. $\text{cont}_G \subseteq V_G \times V_G$ is the *containment relation* induced by the set $C \subseteq E_T$: If $e \in C$ and $v_1 \leq s(e), v_2 \leq t(e)$, then $(v_1, v_2) \in \text{cont}_G$. If $(v_1, v_2), (v_2, v_3) \in \text{cont}_G$, then $(v_1, v_3) \in \text{cont}_G$.

The conditions are said to be *EMF-constraints*.

The second constraint is a monadic second order constraint. Instead of it, we consider the constraint “No containment cycle of length $\leq k$ ” for a fixed natural number k . The resulting constraints, called *EMFk constraints*, are first-order constraints and can be expressed by typed graph constraints [6].

Fact 4 (EMFk -constraints). For the EMFk -constraints, there is a schema of typed graph constraints:

1. At most one container 
2. No containment cycle of length $\leq k$ 
3. No parallel edges 
4. All opposite edges 

where A, B, C, t, t_1, t_2 are node and edge types, respectively, edges without type are arbitrary typed, and \xleftrightarrow{i} denotes a path of containment edges of length i .

The first constraint requires that there are no two different containment edges with a common target. The second constraint requires that there are no loops and no cycles of length $\leq k$. The third constraint requires, that there are no parallel edges of the same type. The fourth constraint requires that, if there is an opposite-edge marking between an A-typed and a B-typed node with type requirement t_1, t_2 , there exists already one edge e_1 with the type t_1 , then an opposite edge with type t_2 in opposite direction should exist.

Fact 5. The instances of EMFk constraints are negative or universal. Every conjunction of EMFk constraint instances is satisfiable. Every sequence of instances of one EMFk constraint is preserving.

Lemma 8 (preservation). Every conjunction of instances of an EMFk constraint is preserving.

Proof. The instances of the first three EMFk constraints are negative; thus, each conjunction of them is preserving. The instances of the fourth EMFk constraint are universal; by induction on the number of constraints, it can be shown that each conjunction is preserving. \square

Let $\text{emfk}_1, \text{emfk}_2$ be conjunctions of EMFk constraints. An *EMFk model repair* program for $\langle \text{emfk}_1, \text{emfk}_2 \rangle$ is an emfk_1 -preserving repair program for emfk_2 . An *EMFk model completion* program is an EMF model repair program for true and for the conjunction of all EMFk constraints. A repair program P for a constraint e is *stable* if, for all transformations $L \Rightarrow_P M$, $L \models e$ implies $L \cong M$, i.e., they do not change the the graph provided the constraint e is satisfied.

Theorem 5 (EMFk model repair & completion). Let emfk_1 is a conjunction of negative EMFk constraints, emfk_2 a conjunction of negative or universal and preserving EMFk constraints, and $\overline{\text{emfk}}$ the conjunction of all EMFk constraints.

1. There is a model-repair program for $\langle \text{emfk}_1, \text{emfk}_2 \rangle$.
2. There is a model-completion program.
3. The EMFk model repair and completion programs are stable.

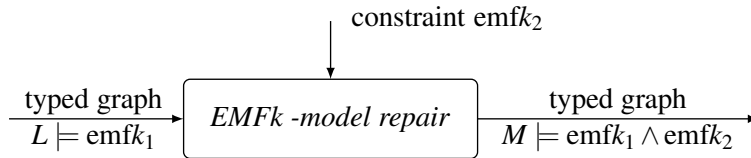


Figure 5: Illustration of EMFk model repair

Proof. 1. By Theorem 1, there are repair program Q_1, Q_2 for $\text{emfk}_1, \text{emfk}_2$, respectively. By Lemma 8, every conjunction emfk_1 and emfk_2 is preserving. Consequently, Lemma 4 can be applied and there is an emfk_1 -preserving repair program $Q_2^{\text{emfk}_1}$ for emfk_2 , and by Fact 3, $\langle Q_1, Q_2^{\text{emfk}_1} \rangle$ is a repair program for $\text{emfk}_1 \wedge \text{emfk}_2$.

2. The statement is an immediate consequence of Theorem 5.1.

3. The statement follows immediately from Construction 3. \square

Remark ((OCL-)Constraints). By the repair results on typed graphs, (model) repair and completion can be done for other constraints satisfying the requirements in Theorem 2, e.g., for first-order (OCL-)constraints.

Inspecting the EMF k repair (completion) program, it turns out that the program deletes and adds an edge, but it does not change the number of nodes.

Fact 6 (preservation of the number of nodes). The application of the EMF k repair (completion) program does not change the number of nodes.

There is a close relationship between EMF k and EMF. For an EMF k constraint emfk , emf denotes the more rigorous EMF constraint requiring no containment cycles and, for an EMF-constraint emf , emfk denotes the weaker EMF k constraint emf requiring no containment cycles of length $\leq k$.

Fact 7 (EMF k -EMF). For typed graphs L of node size $\leq k$, $L \models \text{emfk}$ iff $L \models \text{emf}$.

As a consequence, we obtain the following statement for EMF model repair & completion.

Let $\text{emf}_1, \text{emf}_2$ be conjunctions of EMF constraints. There is *EMF model repair* for a typed graph $L \models \text{emf}_1$ and emf_2 if there is a program P such that, for all transformations $L \Rightarrow_P M$, $M \models \text{emf}_1 \wedge \text{emf}_2$. There is an *EMF model completion* for a typed graph L if there is a program P such that, for all transformations $L \Rightarrow_P M$, M is an EMF-model graph. Model repair and completion for a constraint e are *stable* if, for all typed graphs $L \models e$, all repairs (completions) yield a typed graph isomorphic to L .

Theorem 6 (EMF model repair & completion). Let emf_1 is a conjunction of negative EMF-constraints, emf_2 a conjunction of negative or universal and preserving EMF-constraints, and $\overline{\text{emfk}}$ the conjunction of all EMF k constraints.

1. There is a EMF model repair for all typed graphs $L \models \text{emf}_1$ and $\overline{\text{emfk}}_2$.
2. There is a EMF model completion for all typed graphs L .
3. The EMF model repair and completion for a constraint e are stable for all typed graphs L .

$$\begin{array}{ccc}
 \text{emf}_1 \models L & \xrightarrow{P} & M \models \text{emf}_1 \wedge \text{emf}_2 \\
 \text{if } |V_L| = k \downarrow & & \uparrow \text{if } |V_M| = k \\
 \overline{\text{emfk}}_1 \models L & \xrightarrow{P} & M \models \overline{\text{emfk}}_1 \wedge \overline{\text{emfk}}_2
 \end{array}$$

Figure 6: Relation on emf and emfk

Proof. 1. For a typed graph L of node size k satisfying emf_1 and emf_2 , we take the EMF k repair program P for $\langle \overline{\text{emfk}}_1, \overline{\text{emfk}}_2 \rangle$ and apply it to L . By Fact 7, $L \models \text{emf}_1$ implies $L \models \overline{\text{emfk}}_1$. By Theorem 2, the application of P to L yields a typed graph M satisfying $\overline{\text{emfk}}_1 \wedge \overline{\text{emfk}}_2$. The program does not change the number of nodes, i.e., M is a typed graph with k nodes. By Fact 7, M satisfies $\text{emf}_1 \wedge \text{emf}_2$.

2. For a typed graph L of node size k , we take the EMF k -completion program and apply it to L yielding an EMF k -model graph M , i.e, a typed graph M satisfying $\overline{\text{emfk}}$. The program does not change the number of nodes, i.e., M is a typed graph with k nodes. By Fact 7, M satisfies $\text{emf}_1 \wedge \text{emf}_2$.

3. By Theorem 5, the EMF k -model repair and completion programs are stable, i.e., for all transformations $L \Rightarrow_P M$, $L \models e$ implies $L \cong M$. Applying the programs to a typed graph, the property remains preserved. \square

Remark (Repair of other structures). In this section, the results in Section 3 are applied to meta-modeling: typed graphs are repair w.r.t. EMF constraints. Obviously, typed graphs can also be repaired w.r.t. other constraints, e.g. OCL-graph constraints as considered in [20]. Note that the presented results hold in every \mathcal{M} -adhesive category with \mathcal{E}^l - \mathcal{M} -pair factorization. As a consequence, we can do repair for high-level structures and high-level constraints [4].

Remark (Model generation). In model generation, given a meta-model, one tries to find some (all) instances of the meta-model. Model generation may be seen as a special case of model completion applying the program: For a fixed k , the application of the EMF k model completion program to the empty typed graph yields EMF k model graphs. By Fact 6, every EMF k model graph with node size $\leq k$ is an EMF model graph. In this way, we obtain some instances of the meta-model.

5 Related work

In this section, we present some related concepts on model repair, for which there is a wide variety of different approaches. Recently, there has been a sophisticated survey on different model repair techniques, and a feature-based classification of these approaches, see [11]. In their sense, our approach is *stable* (see Theorem 5.3).

In **Schneider et al. 2019** [22], a logic-based incremental approach to graph repair is presented, generating a sound and complete (upon termination) overview of least changing repairs. The graph repair algorithm takes a graph and a first-order (FO) graph constraint as inputs and returns a set of graph repairs. Given a condition and a graph, they compute a set of symbolic models, which cover the semantics of a graph condition. Both approaches are proven to be **correct**, i.e. the repair (programs) yield to a graph satisfying the condition. The delta-based repair algorithm takes the graph update history explicitly into account, i.e. the approach is **dynamic**. In contrast, our approach is static, i.e., we first construct a repair program, then apply this program to an arbitrary graph. The repair algorithm does not **terminate**, if the repair updates trigger each other ad infinitum. Here, we have constructed terminating repair programs.

In **Biermann et al. 2012** [2], for EMF model transformations, consistent transformations are defined. For a set of rules, they slightly modify them, to get so-called consistent transformation rules. This way, a direct transformation step applied at an EMF model yields an EMF model graph again. In our approach, a direct transformation step leads to typed graphs. We use the repair program to complete the typed graph to an EMF model graph.

In **Nassar et al. 2017** [13], a rule-based approach to support the modeler in automatically trimming and completing EMF models is presented. For that, repair rules are automatically generated from multiplicity constraints imposed by a given meta-model. The rule schemes are carefully designed to preserve the EMF model constraints. One can use the approach in this paper to transform a typed graph to an EMF k model graph, then, one can use the approach of [13], to transform an EMF model graph to an EMF model graph, satisfying additional multiplicity constraints.

In **Nassar et al. 2020** [12], a method to simplify constraint-preserving application conditions is presented. Their simplifications of the application conditions are based on three main concepts: (1) If the elements which are deleted (or added) by a rule are type-disjoint with the types of the constraint, i.e. they share

no types, the application condition simplifies to `true`, (2) For increasing (or decreasing) rules, and positive (or negative) constraints, the application is `true`, (3) For negative constraints $\nexists C$ one may omit the cases where C and the elements created by the rule overlap in at least one element. The simplified application conditions are proven to be logically equivalent to the original application condition. The results are proven to be correct for \mathcal{M} -adhesive categories, and can be used to simplify the application conditions needed for our construction of condition-preserving application conditions (Lemma 1), as well. Furthermore, the EMF model constraints (Fact 4) are simplified by replacing every subcondition violating the no parallel edge, and at most one container constraints with `false`.

In **Kosiol et al. 2020** [9] two notions of consistency as a graduated property are introduced: consistency-sustaining rules do not change the number of violations of a constraint in a graph, and consistency-improving rule reduce the number of violations in a graph. The definition is based on the so-called consistency index, given by the number of constraint violations in a graph, divided by the number of “relevant occurrences” of the constraints in a graph. A transformation is consistency sustaining, if the consistency index for the input graph is equal or less than the resulting graph, and consistency improving if the number of the violations in the resulting graph is smaller than in the input graph. A rule is consistency sustaining, if all transformations are. A rule is consistency improving, if all applications of the rule are consistency sustaining, there exists a graph with constraint violations, and a transformation, such that the number of the violations in the resulting graph is smaller than in the input graph. A rule is strongly consistency improving if all its applications to a graph with constraint violations is consistency improving. In their setting, the rules derived from our approach are strongly consistency improving,

In **Taentzer et al. 2017** [26], a designer can specify a set of so-called change-preserving rules, and a set of edit rules. Each edit rule, which yields to an inconsistency, is then repaired by a set of repair rules. The construction of the repair rules is based on the complement construction. It is shown, that a consistent graph is obtained by the repair program, provided that each repair step is sequentially independent from each following edit step, and each edit step can be repaired. The repaired models are not necessarily as close as possible to the original model.

In **Rabbi et al. 2015** [19], a model completion approach for predicates specified in the Diagrammatic Predicate Framework (DPF) is introduced. For every predicate in the model, they derive a set of completion rules, by constructing the pullback of the instance, the meta-model and the graph of the condition. These rules, applied as long as possible, yields a model which conforms to the predicate. In our approach, the rules are derived from the constraint and the meta-model. In both approaches, the meta-model remains unchanged.

In **Wang 2016** [28], the semantics of the predicates in the DPF is specified as graph constraints, and a model repair approach for these graph constraints is introduced. For constraints of the form $\forall(L, \exists R)$ or $\forall(L, \nexists R)$, repair rules are directly derived from the constraints. The construction is based on the construction of subgraphs of L and R . For the constraint $\forall(L, \exists R)$, for each subgraph B , they derive rules $\langle B \Rightarrow R, \nexists R \rangle$ and $\langle L \Rightarrow B, \nexists R \rangle$. For the constraint $\forall(L, \nexists R)$, rules of the form $\langle L \Rightarrow B \rangle$ are derived. The performance of the approach has been optimized for practical application scenarios. In this work, we have combined the programs for proper conditions to a repair program for conjunctions of proper conditions. The properties of the repaired conditions remain preserved, whenever possible. If this is not possible, we delete the occurrence. As far as we can see, the approach in [28] does not handle conjunctions.

In **Barriga et al 2019** [1], an algorithm for model repair based on EMF is presented, which relies on reinforcement learning. For each error in the model, a so-called Q-table is constructed, storing a weight

for each error, and repair action. This weight indicates how good a repair action is, depending on the repair action and regarding the users preferences. The approach can repair errors provided by the EMF diagnostician. The results are not proven but evaluated using mutation testing.

6 Conclusion

In this paper, we have presented the theory of typed repair programs, applied it to EMF k -and EMF model graph repair.

1. **Typed graph repair.** We have extended our results on graph repair to typed graphs. There are repair programs for a large class of conditions, called *legit* conditions. Application of the repair programs to an arbitrary typed graph yields a typed graph satisfying the condition.
2. **EMF k model repair.** For EMF k constraints, a first-order variant of EMF constraints, we present stable EMF k model repair and completion programs. Application of these programs to any typed graph yields a repaired typed graph and an EMF k model graph, respectively,
3. **EMF model repair.** These results are applied to the EMF world and yield to EMF model repair and completion results.

Further topics may be the following.

1. **Least changng repairs.** Our repair programs induce *repairs* in the sense of Schneider et al. [22]. It would be nice to show that these induced repairs are least changing repairs.
2. **Redirection of edges.** Our repair programs try to preserve items; if this is not possible, they delete items. In Nassar et al. [13], multiplicity constraints are considered. In this context, they use the idea, to redirect an edge instead of deleting it. How this can be included in our approach?
3. **Generalization to attributed type graphs** as e.g. in [20, 16].

Acknowledgements. We are grateful to Annegret Habel, Marius Hubatschek, Jens Kosiol, Okan Özkan, Gabriele Taentzer, and the anonymous reviewers for their helpful comments to this paper.

References

- [1] Angela Barriga, Adrian Rutle & Rogardt Haldal (2019): *Personalized and Automatic Model Repairing using Reinforcement Learning*. In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion*, IEEE, pp. 175–181, doi:10.1109/MODELS-C.2019.00030.
- [2] Enrico Biermann, Claudia Ermel & Gabriele Taentzer (2012): *Formal foundation of consistent EMF model transformations by algebraic graph transformation*. *Software and System Modeling* 11(2), pp. 227–250, doi:10.1007/s10270-011-0199-7.
- [3] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science, Springer, doi:10.1007/3-540-31188-2.
- [4] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers & Fernando Orejas (2014): *\mathcal{M} -Adhesive Transformation Systems with Nested Application Conditions. Part 1: Parallelism, Concurrency and Amalgamation*. *Mathematical Structures in Computer Science* 24, doi:10.1017/S0960129512000357.

- [5] Hartmut Ehrig, Ulrike Golas & Frank Hermann (2010): *Categorical Frameworks for Graph Transformation and HLR Systems based on the DPO Approach*. *Bulletin of the EATCS* 112, pp. 111–121. Available at <http://eatcs.org/beatcs/index.php/beatcs/article/view/158>.
- [6] Annegret Habel & Karl-Heinz Pennemann (2009): *Correctness of High-Level Transformation Systems Relative to Nested Conditions*. *Mathematical Structures in Computer Science* 19, pp. 245–296, doi:10.1017/S0960129508007202.
- [7] Annegret Habel & Christian Sandmann (2018): *Graph Repair by Graph Programs*. In: *Graph Computation Models (GCM 2018)*, *Lecture Notes in Computer Science* 11176, pp. 431–446, doi:10.1007/978-3-030-04771-9_31.
- [8] Reiko Heckel & Gabriele Taentzer (2020): *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer, doi:10.1007/978-3-030-43916-3.
- [9] Jens Kosiol, Daniel Strüber, Gabriele Taentzer & Steffen Zschaler (2020): *Graph Consistency as a Graduated Property - Consistency-Sustaining and -Improving Graph Transformations*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation - 13th International Conference, ICGT, Lecture Notes in Computer Science* 12150, Springer, pp. 239–256, doi:10.1007/978-3-030-51372-6_14.
- [10] Michael Löwe (1993): *Algebraic Approach to Single-Pushout Graph Transformation*. *Theoretical Computer Science* 109, pp. 181–224, doi:10.1016/0304-3975(93)90068-5.
- [11] Nuno Macedo, Jorge Tiago & Alcino Cunha (2017): *A Feature-Based Classification of Model Repair Approaches*. *IEEE Trans. Software Eng.* 43(7), pp. 615–640, doi:10.1109/TSE.2016.2620145.
- [12] Nebras Nassar, Jens Kosiol, Thorsten Arendt & Gabriele Taentzer (2020): *Constructing optimized constraint-preserving application conditions for model transformation rules*. *J. Log. Algebraic Methods Program.* 114, p. 100564, doi:10.1016/j.jlamp.2020.100564.
- [13] Nebras Nassar, Hendrik Radke & Thorsten Arendt (2017): *Rule-Based Repair of EMF Models: An Automated Interactive Approach*. In: *Theory and Practice of Model Transformation (ICMT 2017)*, *Lecture Notes in Computer Science* 10374, pp. 171–181, doi:10.1007/978-3-319-61473-1_12.
- [14] Christian Nentwich, Wolfgang Emmerich & Anthony Finkelstein (2003): *Consistency Management with Repair Actions*. In: *Software Engineering*, IEEE Computer Society, pp. 455–464, doi:10.1109/ICSE.2003.1201223.
- [15] Object Management Group (2014): *Object Constraint Language, Version 2.4, OCL (February 2014)*. Available at <https://www.omg.org/spec/OCL/2.4/>.
- [16] Fernando Orejas & Leen Lambers (2010): *Symbolic Attributed Graphs for Attributed Graph Transformation*. *Electronic Communications of the EASST* 30, doi:10.14279/tuj.eceasst.30.405.
- [17] Karl-Heinz Pennemann (2009): *Development of Correct Graph Transformation Systems*. Ph.D. thesis, Universität Oldenburg.
- [18] Christopher M. Poskitt & Detlef Plump (2013): *Verifying Total Correctness of Graph Programs*. *Electronic Communications of the EASST* 61, doi:10.14279/tuj.eceasst.61.827.
- [19] Fazle Rabbi, Yngve Lamo, Ingrid Chieh Yu & Lars Michael Kristensen (2015): *A Diagrammatic Approach to Model Completion*. In: *Proceedings of the 4th Workshop on the Analysis of Model Transformations, CEUR Workshop Proceedings* 1500, CEUR-WS.org, pp. 56–65. Available at <http://ceur-ws.org/Vol-1500/paper7.pdf>.
- [20] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel & Gabriele Taentzer (2018): *Translating Essential OCL Invariants to Nested Graph Constraints for Generating Instances of Meta-models*. *Science of Computer Programming* 152, pp. 38–62, doi:10.1016/j.scico.2017.08.006.
- [21] Christian Sandmann & Annegret Habel (2019): *Rule-based Graph Repair*. In: *Proceedings Tenth International Workshop on Graph Computation Models, GCM@STAF 2019, EPTCS* 309, pp. 87–104, doi:10.4204/EPTCS.309.5.

- [22] Sven Schneider, Leen Lambers & Fernando Orejas (2019): *A Logic-Based Incremental Approach to Graph Repair*. In: *Fundamental Approaches to Software Engineering - (FASE 2019)*, *Lecture Notes in Computer Science* 11424, pp. 151–167, doi:10.1007/978-3-030-16722-6_9.
- [23] Shane Sendall & Wojtek Kozaczynski (2003): *Model Transformation: The Heart and Soul of Model-Driven Software Development*. *IEEE Software* 20(5), pp. 42–45, doi:10.1109/MS.2003.1231150.
- [24] David Steinberg, Frank Budinsky, Ed Merks & Marcelo Paternostro (2008): *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional.
- [25] Gabriele Taentzer (2012): *Instance Generation from Type Graphs with Arbitrary Multiplicities*. *Electronic Communications of the EASST* 47, doi:10.14279/tuj.eceasst.47.727.
- [26] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo & Adrian Rutle (2017): *Change-Preserving Model Repair*. In: *Fundamental Approaches to Software Engineering (ETAPS 2017)*, *Lecture Notes in Computer Science* 10202, pp. 283–299, doi:10.1007/978-3-662-54494-5_16.
- [27] Guido Wachsmuth (2007): *Metamodel Adaptation and Model Co-adaptation*. In Erik Ernst, editor: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, LNCS 4609*, Springer, pp. 600–624, doi:10.1007/978-3-540-73589-2_28.
- [28] Xiaoliang Wang (2016): *Towards Correct Modelling and Model Transformation in DPF*. Ph.D. thesis, University of Bergen.