Rule-based Graph Repair*

Christian Sandmann, Annegret Habel
Universität Oldenburg
{habel,sandmann}@informatik.uni-oldenburg.de

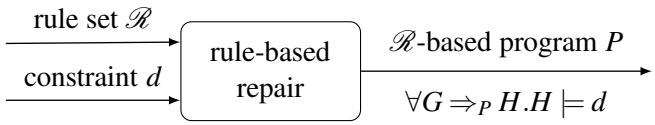
Model repair is an essential topic in model-driven engineering. Since models are suitably formalized as graph-like structures, we consider the problem of rule-based graph repair: Given a rule set and a graph constraint, try to construct a graph program based on the given set of rules, such that the application to any graph yields a graph satisfying the graph constraint. We show the existence of repair programs for specific constraints, and show the existence of rule-based repair programs for specific constraints compatible with the rule set.

1 Introduction

In model-driven software engineering the primary artifacts are models, which have to be consistent w.r.t. a set of constraints (see e.g. [5]). These constraints can be specified by the Object Constraint Language (OCL) [13]. To increase the productivity of software development, it is necessary to automatically detect and resolve inconsistencies arising during the development process, called model repair (see, e.g. [12, 10, 11]). Since models can be represented as graph-like structures [2] and a subset of OCL constraints can be represented as graph conditions [16, 1], we investigate graph repair and rule-based graph repair.

Firstly, the problem of *graph repair* is considered: Given a graph constraint d, we derive repairing sets from the constraint d and try to construct a graph program using this rule set, called *repair program*. The repair program is constructed, such that the application to any graph yields a graph satisfying the graph constraint. Secondly, we consider the problem of *rule-based graph repair*: Given a set of rules $\mathscr{R}$ and a constraint d, try to construct a repair program P based on the rule set $\mathscr{R}$, i.e., we allow to equip the rules of $\mathscr{R}$ with the dangling-edges operator, context, application conditions [6], and interface [14].

Rule-based repair problem



If a graph G is generated by a grammar with rule set $\mathscr{R}$, then, after the application of an $\mathscr{R}$-based program, the result can be generated by the grammar, too. This is interesting in contexts where the language is defined by a grammar, like triple graph grammars [18].

As main results, we show that, (1) there are repair programs for all "proper" conditions, and (2) there are rule-based repair programs for proper conditions provided that the given rule set is compatible with the rule sets of the original program.

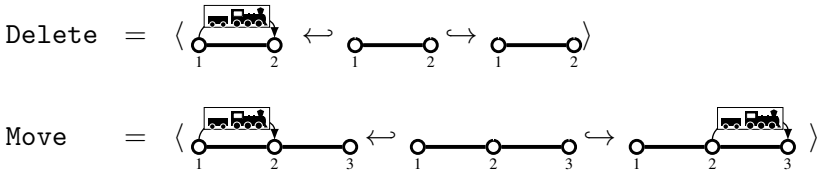
We illustrate our approach by a small railroad system.

*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).

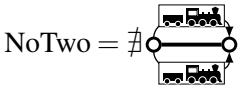
Rachid Echahed and Detlef Plump (Eds.): Tenth International Workshop on Graph Computation Models (GCM 2019)
EPTCS 309, 2019, pp. 87–104, doi:10.4204/EPTCS.309.5

Example 1 (railroad system). The specification of a railroad system is given in terms of graphs, rules (for moving the trains), and conditions. The basic items are waypoints, bi-directional tracks and trains. The static part of the system is given by a directed rail net graph: tracks are modeled by undirected edges (or a pair of directed edges, respectively) and trains are modeled by edges. Source and target nodes of a train edge encode the train's position on the track and the direction of its movement.

The dynamic part of the system is specified by graph transformation rules. The rules model the movement and deletion of trains thereon. Application of the rule `Move` (`Delete`) means to find an occurrence of the left-hand side in the rail net graph and to replace it with the right-hand side of the rule.



In the following, we consider the constraint NoTwo below, meaning that there are no two trains occupying the same piece of track.



One may look for repair programs for the constraint NoTwo based on the rule sets {`Move`}, {`Delete`}, and {`Move`,`Delete`} such that the application to any graph yields a graph satisfying the constraint NoTwo.

The structure of the paper is as follows. In Section 2, we review the definitions of graphs, graph conditions, and graph programs. In Section 3, we introduce repair programs and show that there are repair programs for so-called proper conditions. In Section 4, we introduce rule-based programs, show that there are rule-based programs for transformations, and rule-based repair programs for proper conditions compatible with a rule set. 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 directed, labelled graphs, graph conditions, rules and transformations [4], graph programs [7], and basic transformations [6].

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

Definition 1 (graphs & morphisms). A *(directed, labelled) graph* (over a label alphabet $\mathscr{L}$) is a system $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_{\mathrm{V},G}, \mathrm{l}_{\mathrm{E},G})$ where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*, $\mathrm{s}_G, \mathrm{t}_G \colon \mathrm{E}_G \to \mathrm{V}_G$ are total functions assigning *source* and *target* to each edge, and $\mathrm{l}_{\mathrm{V},G} \colon \mathrm{V}_G \to \mathscr{L}$, $\mathrm{l}_{\mathrm{E},G} \colon \mathrm{E}_G \to \mathscr{L}$ are total labeling functions. If $\mathrm{V}_G = \emptyset$, then G is the *empty graph*, denoted by $\emptyset$. A graph is *unlabelled* if the label alphabet is a singleton. Given graphs G and H, a *(graph) morphism* $g \colon G \to H$ consists of total functions $g_{\mathrm{V}} \colon \mathrm{V}_G \to \mathrm{V}_H$ and $g_{\mathrm{E}} \colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve sources, targets, and labels,

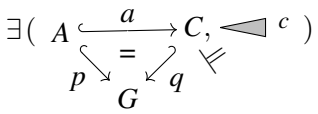
that is, $g_V \circ s_G = s_H \circ g_E$, $g_V \circ t_G = t_H \circ g_E$, $l_{V,G} = l_{V,H} \circ g_V$, $l_{E,G} = l_{E,H} \circ g_E$. The morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is injective and surjective. In the latter case, G and H are *isomorphic*, which is denoted by $G \cong H$. An injective morphism $g\colon G \hookrightarrow H$ is an *inclusion morphism* if $g_V(v) = v$ and $g_E(e) = e$ for all $v \in V_G$ and all $e \in E_G$.

Convention. Drawing a graph, nodes are drawn as circles with their labels (if existent) inside, and edges are drawn as arrows with their labels (if existent) placed next to them. Arbitrary graph morphisms are drawn by usual arrows $\to$, injective graph morphisms are distinguished by $\hookrightarrow$.

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 [6].

Definition 2 (graph conditions). A *(graph) condition* over a graph A is of the form (a) `true` or $\exists(a,c)$ where $a\colon A \hookrightarrow C$ is a proper inclusion morphism[1] 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[2]) over A, $\wedge_{i\in I} c_i$ is a condition over A. Conditions over the empty graph $\emptyset$ are called *constraints*. In the context of rules, conditions are called *application conditions*. Conditions built by (a) and (b) are called *linear*.

Any injective morphism $p\colon A \hookrightarrow G$ *satisfies* `true`. An injective morphism p *satisfies* $\exists(a,c)$ with $a\colon A \hookrightarrow C$ if there exists an injective morphism $q\colon C \hookrightarrow G$ such that $q \circ a = p$ and q satisfies c.



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

Two conditions c and c' over A are *equivalent*, denoted by $c \equiv c'$, if for all graphs G and all injective morphisms $p\colon A \hookrightarrow G$, $p \models c$ iff $p \models c'$. A condition c *implies* a condition c', denoted by $c \Rightarrow c'$, if for all graphs and all injective morphisms $p\colon A \hookrightarrow G$, $p \models c$ implies $p \models c'$.

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

Example 2. The expression $\neg\exists(\emptyset \hookrightarrow$  $, \neg\exists($  $\hookrightarrow$  $,\texttt{true}) \vee \neg\exists($ $\hookrightarrow$ $,\texttt{true}))$ is a constraint according to Definition 2, written in compact form as $\forall($ $, \exists($ $) \wedge \exists($ $))$ meaning that, for every node, there exists a real [3] outgoing and a real incoming edge.

[1] Without loss of generality, we may assume that for all inclusion morphisms $a\colon A \hookrightarrow C$ in the condition, A is a proper subgraph of C.

[2] In this paper, we consider graph conditions with finite index sets.

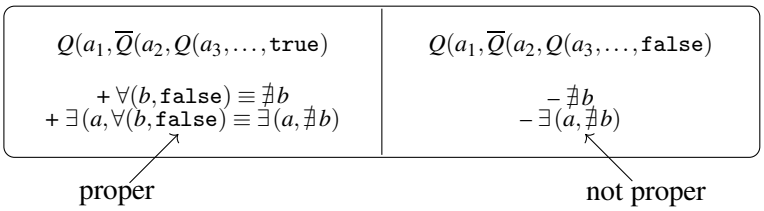
[3] An edge is said to be *real*, if it is not a loop.

Fact 1 (equivalences [14]). There are the following equivalences:

$$\begin{array}{lcllcl}
\exists(x,\texttt{true}) & \equiv & \exists x & \forall(x,\texttt{true}) & \equiv & \texttt{true} \\
\exists(x,\texttt{false}) & \equiv & \texttt{false} & \forall(x,\texttt{false}) & \equiv & \nexists x \\
\forall(x,\exists(y,\texttt{false})) & \equiv & \forall(x,\texttt{false}) \equiv \nexists x & \exists(x,\forall(y,\texttt{false})) & \equiv & \exists(x,\nexists y)
\end{array}$$

To simplify our reasoning, the repair program operates on a subset of conditions in normal form, so-called conditions with alternating quantifiers.

Definition 3 (alternating quantifiers, proper and basic conditions). A linear condition of the form $\mathrm{Q}(a_1,\overline{\mathrm{Q}}(a_2,\mathrm{Q}(a_3,\ldots)))$ with $\mathrm{Q}\in\{\forall,\exists\}$, $\overline{\forall}=\exists$, $\overline{\exists}=\forall$ ending with `true` or `false` is a *condition with alternating quantifiers (ANF)*. Such a condition in ANF is *proper* if it ends with a condition $\exists(b,\texttt{true})\equiv\exists b$ or it is a condition of the form $\exists(a,\forall(b,\texttt{false}))\equiv\exists(a,\nexists b)$ or $\forall(b,\texttt{false})\equiv\nexists b$. A proper condition is *basic* if it is of the form $\exists b$ or $\nexists b$.



Example 3. The linear conditions $\forall($  $,\exists($  $,\texttt{true}))$ and $\forall($ $,\exists($ $,\texttt{false}))$ are conditions with alternating quantifiers. $\forall($  $,\exists($ $,\texttt{true}))$ and $\exists($ $,\forall($ $,\exists($ $,\texttt{true})))$ are proper. Moreover, $\forall($  $,\texttt{false})\equiv\nexists($ $)$ is proper. The linear condition $\forall($ $,\exists($ $,\forall($ $,\texttt{false})))\equiv\forall($ $,\exists($ $,\nexists($ $)))$ is non-proper.

By a normal form result for conditions [14], we obtain a normal form result for linear conditions saying that every linear condition effectively can be transformed into an equivalent condition with alternating quantifiers.

Fact 2 (normal form). For every linear condition, there exists an equivalent condition with alternating quantifiers.

Proof. By a conjunctive normal form result in [14], every condition can be effectively transformed equivalent condition in normal form. The application of the rule $\nexists(a,\neg c)\equiv\forall(a,c)$ as long as possible yields an equivalent condition with alternating quantifiers. □

By definition, proper conditions are satisfiable.

Fact 3 (proper conditions are satisfiable). Every proper condition is satisfiable.

Proof. By Definition 3, a proper condition is `true`, ends with a condition of the form $\exists(x,\texttt{true})\equiv\exists x$, $\forall(x,\texttt{true})\equiv\texttt{true}$, or is of the form $\nexists b$ or $\exists(a,\nexists b)$ and b is not an isomorphism. Thus, it is satisfiable.□

Fact 4 (non-proper satisfiable conditions). There are non-proper conditions that are satisfiable.

Proof. The non-proper condition $\forall($  $,\exists($  $,\forall($ $,\texttt{false})))$ can be transformed into a proper one: $\forall($  $,\exists($ $,\nexists($ $)))\equiv\forall($ $,\texttt{false})\equiv\nexists$. By Fact 3, the condition is satisfiable. □

Plain rules are specified by a pair of injective graph morphisms. They may be equipped with context, application conditions, and interfaces. For restricting the applicability of rules, the rules are equipped with a left application condition. By extending the rules with a context, it is possible to require an application condition over an extended left-hand side (see Example 12). By the interfaces, it becomes possible to hand over information between the transformation steps.

Definition 4 (rules and transformations). A *plain rule* $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ consists of two inclusion morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$. The rule p *equipped with context* $K \hookrightarrow K'$ is the rule $\langle L' \hookleftarrow K' \hookrightarrow R' \rangle$ where L' and R' are the pushout objects in the diagrams (1) and (2) below.

$$\begin{array}{ccccc} L & \hookleftarrow & K & \hookrightarrow & R \\ \vdots & (1) & \downarrow & (2) & \vdots \\ L' & \hookleftarrow & K' & \hookrightarrow & R' \end{array}$$

A *rule* $\rho = \langle x, p, \mathrm{ac}, y \rangle$ *with interfaces* X and Y consists of a plain rule $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ with left application condition ac and two injective morphisms $x\colon X \hookrightarrow L$, $y\colon Y \hookrightarrow R$, called the *(left and right) interface morphisms*. If both interfaces are empty, i.e., the domains of the interface morphisms are empty, we write $\rho = \langle p, \mathrm{ac} \rangle$. If additionally $\mathrm{ac} = \mathtt{true}$, we write $\rho = \langle p \rangle$ or short p. A *direct transformation* $G \Rightarrow_{\rho,g,h,i} H$ or short $G \Rightarrow_\rho H$ from G to H applying ρ at $g\colon X \hookrightarrow G$ consists of the following steps:

(1) Select a match $g'\colon L \hookrightarrow G$ such that $g = g' \circ x$ and $g' \models \mathrm{ac}$.

(2) Apply the plain rule[4] p at g' (possibly) yielding a comatch $h'\colon R \hookrightarrow H$.

(3) Unselect $h\colon Y \hookrightarrow H$, i.e., define $h = h' \circ y$.

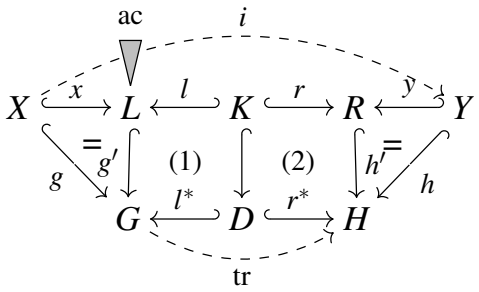


Figure 1: A direct transformation

A triple $\langle g, h, i \rangle$ with partial[5] morphism $i = y^{-1} \circ r \circ l^{-1} \circ x$ (called *interface relation*) is in the semantics of ρ, denoted by $[\![\rho]\!]$, if there is an injective morphism $g'\colon L \hookrightarrow G$ such that $g = g' \circ x$ and $g' \models \mathrm{ac}$, $G \Rightarrow_{p,g',h'} H$, and $h = h' \circ y$. We write $G \Rightarrow_{\rho,g,h,i} H$ or short $G \Rightarrow_\rho H$. Given graphs G, H and a finite set $\mathscr{R}$ of rules, G *derives* H by $\mathscr{R}$ if $G \cong H$ or there is a sequence of direct transformations $G = G_0 \Rightarrow_{\rho_1,g_1,h_1} G_1 \Rightarrow_{\rho_2,h_1,h_2} \ldots \Rightarrow_{\rho_n,h_{n-1},h_n} G_n = H$ with $\rho_1, \ldots, \rho_n \in \mathscr{R}$. In this case, we write $G \Rightarrow^*_{\mathscr{R}} H$ or just $G \Rightarrow^* H$.

Notation. If both interfaces of $\rho = \langle x, p, \mathrm{ac}, y \rangle$ are empty, we write $\rho = \langle p, \mathrm{ac} \rangle$. If additionally $\mathrm{ac} = \mathtt{true}$, we write $\rho = \langle p \rangle$ or short p. A plain rule $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ sometimes is denoted by $L \Rightarrow R$ where

[4] The application of a plain rule is as in the double-pushout approach [4].

[5] A *partial* morphism $i\colon X \rightharpoonup Y$ is an injective morphism $X' \hookrightarrow Y$ such that $X' \subseteq X$.

indexes in L and R refer to the corresponding nodes. Moreover, $\mathtt{Sel}(x,\mathrm{ac})$ and $\mathtt{Uns}(x)$ denote the rules $\langle x,\mathrm{id},\mathrm{ac}\rangle$ (selection of elements) and $\langle \mathrm{id},\mathrm{y}\rangle$ (unselection of selected elements), respectively, where id denotes the identical rule $\langle L \hookleftarrow L \hookrightarrow L\rangle$. Additionally, $\mathtt{Sel}(x)$ abbreviates $\mathtt{Sel}(x,\mathtt{true})$.

Example 4. Consider the rule $\rho = \langle x,p,y\rangle$ with the plain rule $p = \langle$  $\rangle$, and the interface morphisms x: , y:  (see Figure 2).

Figure 2: A direct example transformation

Each injective morphism g:  fixes a node in the host graph. In general, the morphism g restricts the allowed matches g' from the left-hand-side into the host graph by $g' = g \circ x$. The plain rule is applied at g' according the double-pushout approach yielding the comatch h':  . Defining $h = h' \circ y$, we fix the node 1 for the next rule application. It says that at this position (and no other) the rule shall be applied.

With every transformation $t\colon G \Rightarrow^* H$, a partial track morphism can be associated that "follows" the items of G through the transformation: this morphism is undefined for all items in G that are removed by t, and maps all other items to the corresponding items in H.

Definition 5 (track morphism [15]). The *track morphism* $\mathrm{tr}_{G\Rightarrow H}$ from G to H is the partial morphism defined by $\mathrm{tr}_{G\Rightarrow H}(x) = r^*(l^{*-1}(x))$ if $x \in D$ and *undefined* otherwise, where the morphisms $l^*\colon D \hookrightarrow G$ and $r^*\colon D \hookrightarrow H$ are the induced morphisms of $l\colon K \hookrightarrow L$ and $r\colon K \hookrightarrow R$, respectively (see Figure 1). Given a transformation $G \Rightarrow^* H$, $\mathrm{tr}_{G\Rightarrow^* H}$ is defined by induction on the length of the transformation: $\mathrm{tr}_{G\Rightarrow^* H} = \mathrm{iso}$ for an isomorphism $\mathrm{iso}\colon G \to H$ and $\mathrm{tr}_{G\Rightarrow^* H} = \mathrm{tr}_{G'\Rightarrow H} \circ \mathrm{tr}_{G\Rightarrow^* G'}$ for $G \Rightarrow^+ H = G \Rightarrow^* G' \Rightarrow H$.

Example 5. For the direct transformation $t\colon G \Rightarrow_\rho H$ in Example 4, the track morphism is tr_t:  . For the direct inverse transformation t', $\mathrm{tr}_{t'}$: is partial.

Graph programs are made of sets of rules with interface, non-deterministic choice, sequential composition, as-long-as possible iteration, and the try-statement.

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

(1) Every rule ρ with interface X (and Y) is in $\mathrm{Prog}(X)$.
(2) If $P,Q \in \mathrm{Prog}(X)$, then $\{P,Q\}$ is in $\mathrm{Prog}(X)$ (nondeterministic choice).
(3) If $P \in \mathrm{Prog}(X)$ and $Q \in \mathrm{Prog}(Y)$, then $\langle P;Q\rangle \in \mathrm{Prog}(X)$ (sequential composition).
(4) If $P \in \mathrm{Prog}(X)$, then $P\downarrow$, and $\mathtt{try}\ P$ are in $\mathrm{Prog}(X)$ (iteration & try).

The *semantics* of a program P with interface X, denoted by $[\![P]\!]$, is a set of triples such that, for all $\langle g,h,i\rangle \in [\![P]\!]$, $X = \mathrm{dom}(g) = \mathrm{dom}(i)$[6] and $\mathrm{dom}(h) = \mathrm{ran}(i)$, and is defined as follows:

$$\begin{array}{lll}
(1) & [\![\rho]\!] & \text{as in Definition 4} \\
(2) & [\![\{P,Q\}]\!] & = [\![P]\!] \cup [\![Q]\!] \\
(3) & [\![\langle P;Q\rangle]\!] & = \{\langle g_1,h_2,i_2\circ i_1\rangle \mid \langle g_1,h_1,i_1\rangle\in[\![P]\!], \langle g_2,h_2,i_2\rangle\in[\![Q]\!] \text{ and } h_1 = g_2\} \\
(4) & [\![P\downarrow]\!] & = \{\langle g,h,\mathrm{id}\rangle \in P^* \mid \nexists h'.\langle h,h',\mathrm{id}\rangle \in [\![\mathtt{Fix}(P)]\!]\} \\
 & [\![\mathtt{try}\ P]\!] & = \{\langle g,h,i\rangle \mid \langle g,h,i\rangle \in [\![P]\!]\} \cup \{\langle g,g,\mathrm{id}\rangle \mid \nexists h.\langle g,h,i\rangle \in [\![P]\!]\}
\end{array}$$

where $P^* = \bigcup_{j=0}^{\infty} P^j$ with $P^0 = \mathtt{Skip}$, $P^j = \langle \mathtt{Fix}(P); P^{j-1}\rangle$ for $j > 0$ and $[\![\mathtt{Fix}(P)]\!] = \{\langle g,h\circ i,\mathrm{id}\rangle \mid \langle g,h,i\rangle \in [\![P]\!]\}$. Two programs P,P' are *equivalent*, denoted $P \equiv P'$, if $[\![P]\!] = [\![P']\!]$. A program P is *terminating* if the relation $\to$ is terminating.

The statement `Skip` is the identity element $\mathtt{Sel}(\mathrm{id},\mathtt{true})$ of sequential composition.

Example 6. Consider a slightly modified example as in Example 4. For restricting the applicability of the plain rule $\mathtt{AddEdge} = \langle \bullet_1 \hookleftarrow \bullet_1 \hookrightarrow \bullet_1\!\to\!\bullet_2 \rangle$ to a fixed node, the rule is equipped with a right interface $y_1\colon \bullet_1 \hookrightarrow \bullet_1\!\to\!\bullet_2$ yielding the rule $\mathtt{AddEdge}_1 = \langle \mathtt{AddEdge}, y_1\rangle$ as well as with a left interface $x_2\colon \bullet_1 \hookrightarrow \bullet_1$ yielding the rule $\mathtt{AddEdge}_2 = \langle \mathtt{AddEdge}, x_2\rangle$. By Definitions 4 and 6, $h'_1 \circ y_1 = h_1 = g_2 = g'_2 \circ x_2$, i.e., the middle diagram commutes (see Figure 3).

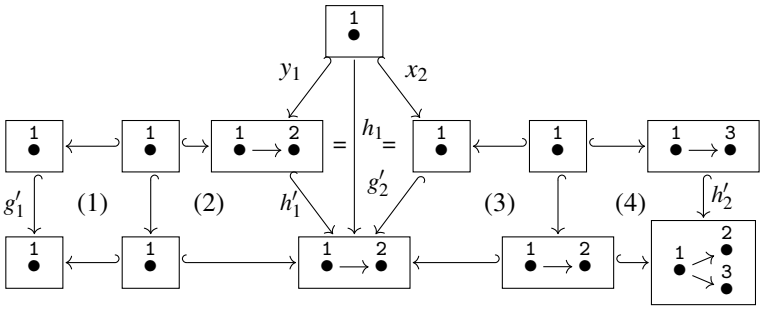
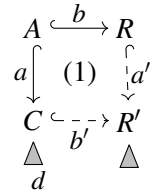


Figure 3: A sequence of direct transformations

The construction Shift "shifts" existential conditions over morphisms into a disjunction of existential application conditions.

Lemma 1 (Shift **[6]**). There is a construction Shift, such that the following holds. Let d be condition over A and $b\colon A \hookrightarrow R, n\colon R \hookrightarrow H$. Then $n \circ b \models d \iff n \models \mathrm{Shift}(b,d)$.

Construction 1. For rules ρ with plain rule $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$, the construction is as follows.



$\mathrm{Shift}(b,\mathtt{true}) := \mathtt{true}$.
$\mathrm{Shift}(b,\exists(a,d)) := \bigvee_{(a',b')\in\mathscr{F}} \exists(a',\mathrm{Shift}(b',d))$ where
$\mathscr{F} = \{(a',b') \mid b'\circ a = a'\circ b,\ a',b' \text{ inj, } (a',b') \text{ jointly surjective}^7\}$
$\mathrm{Shift}(b,\neg d) := \neg\mathrm{Shift}(b,d)$, $\mathrm{Shift}(b,\wedge_{i\in I} d_i) := \wedge_{i\in I}\mathrm{Shift}(b,d_i)$.

[6] For a partial morphism i, $\mathrm{dom}(i)$ and $\mathrm{ran}(i)$ denote the domain and codomain of i, respectively.

[7] A pair (a',b') 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 $b'(z) = x$.

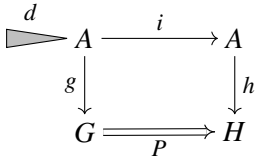
Example 7 (Shift)**.** The application of Shift to the injective morphism $b\colon \emptyset \hookrightarrow$ and the condition $d = \exists(\emptyset \hookrightarrow$  $)$ yields the condition $\mathrm{Shift}(b,d) = \exists($  $\hookrightarrow$ $) \vee \exists($ $\hookrightarrow$ $)$. The application of Shift to b and of the condition $d' = \exists(\emptyset \hookrightarrow$ $, \nexists$ $)$ over b yields the condition $\mathrm{Shift}(b,d') = \exists($  $\hookrightarrow$ $, \nexists$ $) \vee \exists($ $\hookrightarrow$ $, \nexists$ $)$.

3 Graph repair

In this section, we define repair programs and look for repair programs for graph conditions.

A repair program for a constraint is a program such that, for every application to a graph, the resulting graph satisfies the constraint. More generally, a repair program for a condition over a graph A is a program P with interface A such that for every triple $\langle g,h,i\rangle$ in the semantics of P, the composition of the interface relation i and the comatch h satisfies the condition.

Definition 7 (repair programs). A program P is a *repair program* for a constraint d if, for all transformations $G \Rightarrow_P H$, $H \models d$. An A-preserving program [8] P is a *repair program* for a condition d over A, if, for all triples $\langle g,h,i\rangle \in [\![P]\!]$, $h \circ i \models d$.



Example 8. For the condition $c = \exists($ $\hookrightarrow$ $)$, the $\bullet$ -preserving program $P_c = \mathtt{try}\ \mathscr{R}$ is a repair program for c, where $\mathscr{R} = \langle x,$ $\Rightarrow$ $, \nexists$ $, y\rangle$, with the interface morphisms $x\colon$ $\hookrightarrow$ $, y\colon$ $\hookleftarrow$. For the constraint $d = \forall($ $\bullet$ $, \exists$ $)$, meaning that every node has a loop, the program $\langle \mathtt{Sel}(\emptyset \hookrightarrow$ $); P_c; \mathtt{Uns}($ $\hookleftarrow \emptyset)\rangle \downarrow$ is a repair program for d.

Remark. A program for a condition is *destructive*, if it deletes the input graph and creates a graph satisfying the condition from the empty graph. In general, destructive programs are no repair programs for d over $A \neq \emptyset$, because it is not A-preserving.

The most significant point are the repair programs for the basic conditions $\exists a$ and $\nexists a$. Whenever we have repairing sets, we obtain a repair program for proper conditions.

Definition 8 (repairing sets). Let $a\colon A \hookrightarrow C$ with $A \subset C$. An A-set $\mathscr{R}_a$ is *repairing* for $\exists a$ if $\mathtt{try}\ \mathscr{R}_a$ is a repair program for $\exists a$. An A-set $\mathscr{S}_a$ is *repairing* for $\nexists a$ if $\mathscr{S}'_a\downarrow$ (see Definition 9) is a repair program for $\nexists a$.

Example 9. For the condition c from Example 8, the repairing set is $\mathscr{R}$.

[7] For a rule $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$, $p^{-1} = \langle R \hookleftarrow K \hookrightarrow L\rangle$ denotes the *inverse* rule. For $L' \Rightarrow_p R'$ with intermediate graph K', $\langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ is the *derived* rule.

[8] A program is *A-preserving* if the dependency relation i of the program is total. If, additionally, the codomain of i is A, the program is a *program with interfaces A* or short *A-program.* For a rule set with interfaces A, we speak of *A-set*.

Definition 9 (The dangling-edges operator). For node-deleting rules ρ, the dangling condition[9] may be not satisfied. In this case, we consider the program ρ' that fixes a match for the rule, deletes the dangling edges, and afterwards applies the rule at the match. The program corresponds with the SPO-way of rewriting [9]. The proceeding can be extended to sets of rules: For a rule set $\mathscr{S}$, $\mathscr{S}' = \{\rho' \mid \rho \in \mathscr{S}\}$.

In the following, we show that, for basic conditions $\exists(A \hookrightarrow C)$ and $\nexists(A \hookrightarrow C)$ over A, there are repairing A-sets $\mathscr{R}_a$ and $\mathscr{S}_a$, respectively. The rules in $\mathscr{R}_a$ are increasing and of the form $B \Rightarrow C$ where $A \subseteq B \subset C$ and an application condition requiring that no larger subgraph B' of C occurs and the shifted condition $\nexists a$ is satisfied. By the application condition, each rule can only be applied iff the condition is not satisfied and no other rule whose left-hand side includes B and is larger can be applied. The rules in $\mathscr{S}_a$ are decreasing and 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. The rule set $\mathscr{R}_a$ can be used, e.g., for the repair program of the condition $\forall(x, \exists a)$, the rule set $\mathscr{S}_a$ for the condition $\exists(x, \nexists a)$ (see Construction 3).

Lemma 2 (basic repair). For basic conditions over A, there are repairing sets with interfaces A.

There are several repairing sets for a basic condition: We present two examples of repairing sets. The first one is quite intuitive, but, in general does not lead to a terminating and maximally preserving repair program. The second one is more complicated, but yields a terminating and maximally preserving repair program.

Construction 2. For $d = \exists a$ ($\nexists a$) with $a\colon A \hookrightarrow C$, $A \subset C$, the sets $\mathscr{R}_a$ and $\mathscr{S}_a$ are constructed as follows.

(1) $\mathscr{R}_a = \{\langle \mathrm{id}_A, A \Rightarrow C, a\rangle\}$ and $\mathscr{S}_a = \{\langle a, C \Rightarrow A, \mathrm{id}_A\rangle\}$

(2) $\mathscr{R}_a = \{\langle b, B \Rightarrow C, \mathrm{ac} \wedge \mathrm{ac}_B, a\rangle \mid A \hookrightarrow^b B \subset C\}$ and $\mathscr{S}_a = \{\langle a, C \Rightarrow B, b\rangle \mid A \hookrightarrow^b B \subset C$ and (*)$\}$
where $\mathrm{ac} = \mathrm{Shift}(A \hookrightarrow B, \nexists a)$, $\mathrm{ac}_B = \bigwedge_{B'} \nexists B'$, $\bigwedge_{B'}$ ranges over B' with $B \subset B' \subseteq C$, and
(*) `if` $\mathrm{E}_C \supset \mathrm{E}_B$ `then` $|\mathrm{V}_C| = |\mathrm{V}_B|, |\mathrm{E}_C| = |\mathrm{E}_B| + 1$ `else` $|\mathrm{V}_C| = |\mathrm{V}_B| + 1$.

Proof. (1) Let $\langle g, h, i\rangle \in [\![\texttt{try}\ \mathscr{R}_a]\!]$. If $g \models \nexists a$, then the rule $\langle \mathrm{id}_A, A \Rightarrow C, a\rangle$ in $\mathscr{R}_a$ is applicable and $h \circ i \models \exists a$. If $g \models \exists a$, then, by the semantics of `try` , $g = h \circ i \models \exists a$. Thus, `try` $\mathscr{R}_a$ is a repair program for $\exists a$. Let $\langle g, h, i\rangle \in [\![\mathscr{S}'_a{\downarrow}]\!]$. By the semantics of $\downarrow$, $\mathscr{S}'_a$ is not applicable to the domain of $h \circ i$, i.e., $h \circ i \models \nexists a$. Thus, $\mathscr{S}'_a{\downarrow}$ is a repair program for $\nexists a$. For Construction (2), see the proof [8, Thm 1]. □

Example 10. 1. Consider the condition $d = \exists a$, with $a\colon$  . By Construction (1), the rule $\rho = \langle x, p, y\rangle$ with the plain rule $p = \langle$ $\rangle$ and the interface morphisms $x\colon$  , $y\colon$  constitutes the repairing set for $\exists a$. By Construction (2), we obtain a repairing set $\mathscr{R}_a$ for $\exists a$.

$$\mathscr{R}_a = \begin{cases} \rho_1 = \langle x_1,\ \bullet_1 \quad \Rightarrow \quad \bullet_1 \to \bullet\ ,\nexists\ \bullet_1 \quad \bullet\ ,y_1\rangle \\ \rho_2 = \langle x_2,\ \bullet_1 \quad \bullet_2 \quad \Rightarrow \quad \bullet_1 \to \bullet_2\ ,\nexists\ \bullet_1 \to \bullet_2\ \wedge \nexists\ \bullet_1 \to \bullet \quad \bullet_2\ ,y_2\rangle \end{cases}$$

where the interface morphisms x_i, y_i can be unambiguously inferred. The first rule requires a node and attaches a node and a real outgoing edge, provided that there do not exist two nodes. The second rule

[9] The *dangling condition* for a rule $\rho = \langle L \hookleftarrow K \hookrightarrow R\rangle$ and an injective morphism $g\colon L \hookrightarrow G$ requires: "No edge in $G - g(L)$ is incident to a node in $g(L-K)$".

requires two nodes and attaches a real outgoing edge provided there is no real outgoing edge from the image of node 1 to the image of node 2, and no real outgoing edge at the image of node 1. The rule set $\mathscr{R}_a$ can be used, e.g., for a repair program $\texttt{try}\ \mathscr{R}_a$ for the condition $\exists a$.

2. Consider the condition $\nexists b$, with $b\colon \bullet_1 \hookrightarrow \bullet_1 \circlearrowright$. By Construction (1), the rule set $\{\rho\}$ with $\rho = \langle x,\ \bullet_1 \circlearrowright \Rightarrow \bullet_1\ , y\rangle$ constitutes the repairing set for $\nexists b$, with interfaces $\bullet_1$. By Construction (2), the rule set $\mathscr{S}_b = \langle x,\ \bullet_1 \circlearrowright_2 \Rightarrow \bullet_1 \to \bullet_2\ , y\rangle$ constitutes the repairing set for $\nexists b$, with interfaces $\bullet_1$. The rule set $\mathscr{S}_b$ can be used, e.g., for a repair program $\mathscr{S}'_b\downarrow$ for the condition $\nexists(\bullet_1 \hookrightarrow \bullet_1 \circlearrowright)$ (see Construction 3).

Fact 5 (compositions of repairing sets). If $\mathscr{R}_a, \mathscr{R}'_a$ are repairing sets for a basic condition d, then $\mathscr{R}_a \cup \mathscr{R}'_a$ is a repairing set for d. If $\mathscr{R}_a, \mathscr{R}_c$ are repairing sets for $\exists a$ and $\exists c$, respectively, and $a = c \circ b$, then $\mathscr{R}_a \cup \mathscr{R}_{ca}$ is a repairing set for $\exists a$, where $\mathscr{R}_{ca} = \{\langle b, p, y\rangle \mid \langle p, y\rangle \in \mathscr{R}_c\}$. If $\mathscr{S}_a, \mathscr{S}_c$ are repairing sets for $\nexists a$ and $\nexists c$, respectively, and $a = c \circ b$, then $\mathscr{S}_a \cup \mathscr{S}_{ca}$ is a repairing set for $\nexists a$, where $\mathscr{S}_{ca} = \{\langle b, p, y\rangle \mid \langle p, y\rangle \in \mathscr{S}_c\}$.

Proof. Straightforward. □

For proper conditions, a repair program can be constructed.

Theorem 1 (repair). For proper conditions, repair programs can be constructed.

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

(1) For $d = \texttt{true}$, $P_d = \texttt{Skip}$.

(2) For $d = \exists a$, $P_d = \texttt{try}\ \mathscr{R}_a$.

(3) For $d = \nexists a$, $P_d = \mathscr{S}'_a\downarrow$.

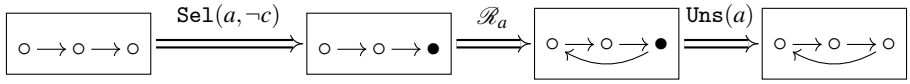
(4) For $d = \exists(a,c)$, $P_d = P_{\exists a}; \langle \texttt{Sel}(a); P_c; \texttt{Uns}(a)\rangle$.

(5) For $d = \forall(a,c)$, $P_d = \langle \texttt{Sel}(a, \neg c); P_c; \texttt{Uns}(a)\rangle\downarrow$

where $a\colon A \hookrightarrow C$ with $A \subset C$, $\mathscr{R}_a$ and $\mathscr{S}_a$ are repairing A-sets, and P_c is a repair program for c with interfaces C.

Example 11. 1. For the constraint $d = \exists(\bullet_1, \nexists\ \bullet_1 \circlearrowright)$ meaning there exists a node without 2-cycle, i.e., two real edges in opposite direction, $P_d = \langle \texttt{try}\ \mathscr{R}_a; \langle \texttt{Sel}(a); \mathscr{S}'_b\downarrow; \texttt{Uns}(a)\rangle\rangle$ where $a\colon \emptyset \hookrightarrow \bullet_1$, $b\colon \bullet_1 \hookrightarrow \bullet_1 \circlearrowright$, and $\mathscr{S}_b$ is the repairing set from Example 10. The program checks whether there exists a node, and if not, it creates one. It selects a node and, if there are two edges in opposite directions, it deletes one. The check of existence is done one time, the deletion as long as possible.

2. For the constraint $d = \forall(\bullet_1, \exists\ \bullet_1 \to \bullet)$, meaning that, for every node, there exists a real outgoing edge, $P_d = \langle \texttt{Sel}(a, \neg c); P_c; \texttt{Uns}(a)\rangle\downarrow$, is a repair program for d, where $a\colon \emptyset \hookrightarrow \bullet_1$, $P_c = \texttt{try}\ \mathscr{R}_a$ is the repair program for $c = \exists(\bullet_1 \hookrightarrow \bullet_1 \to \bullet)$, and $\mathscr{R}_a$ is the repairing set from Example 10. The repair program selects a node without a real outgoing edge, e.g. the third node from left (see below), applies the rule, and unselects the selected part. Afterwards all nodes possess a real outgoing edge.



Proof (of Theorem 1). By induction on the structure of the condition. Let d be a proper condition and P_d the program in Construction 3. (1) Let $d = \texttt{true}$. For all triples $\langle g,h,i\rangle \in [\![\texttt{Skip}]\!]$, $h \circ i \models \texttt{true}$, i.e., `Skip` is a repair program for `true`. For (2) and (3) see Lemma 2.

(4) Let $\langle g,h,i\rangle \in [\![P_{\exists a};\langle \texttt{Sel}(a);P_c;\texttt{Uns}(a)\rangle]\!]$ (see Figure 4, left). We show that $h \models \exists(a,c)$, i.e., there is some injective morphism $q\colon C \hookrightarrow H$ such that $p = q \circ a$ and $q \models c$. Let $\langle g,h_1,i_1\rangle \in [\![P_{\exists a}]\!]$ with $h_1 \circ i_1 \models \exists a$, $\langle h_1,h_2,a\rangle \in [\![\texttt{Sel}(a)]\!]$ with $h_1 = h_2 \circ a$, $\langle h_2,h_3,i_2\rangle \in [\![P_c]\!]$ with $h_3 \circ i_2 \models c$, $\langle h_3,h,a^{-1}\rangle \in [\![\texttt{Uns}(a)]\!]$ with $h = h_3 \circ a$. Choose $q = h_3$. Then $h = h_3 \circ a = q \circ a$ and, since P_d is A-preserving, $q = h_3 = h_3 \circ i_2 \models c$, i.e., $h \models \exists(a,c)$. (For A-preserving programs, the interface relation i_2 total. Without loss of generality, it is an inclusion.)

(5) Let $\langle g,h,i\rangle \in [\![\langle \texttt{Sel}(a,\neg c);P_c;\texttt{Uns}(a)\rangle\!\downarrow]\!]$. We show that $h \models \forall(a,c) = \neg\exists(a,\neg c)$. (see Figure 4, right.) By the semantics of $\downarrow$, the program $\langle \texttt{Sel}(a,\neg c);P_c;\texttt{Uns}(a)\rangle$ is not applicable to the domain of h_1. Then there is an injective morphism h such that $h = h_1 \circ a$ and $h \models \neg c$, i.e., $h \models \neg\exists(a,\neg c) = \forall(a,c)$. □

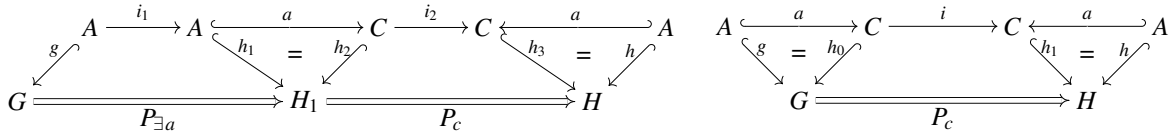


Figure 4: Illustration of the proof

In the following, we look for properties of the constructed repair programs. Whenever a condition graph requires the non-existence (existence) of a certain subgraph, there is no non-deleting (non-adding) repair program that repairs all graphs. Therefore, we look for minimally deleting or maximally preserving repair programs.

Definition 10 (maximally preserving repair). A repair program P_d for a proper condition d is *maximally preserving*, if, for all transformations $t\colon G \Rightarrow_{P_d,g,h} H$,

$$\texttt{pres}(P_d,t) \geq \text{size}(G) - \Delta(g,d)$$

where, for a transformation t via P_d, $\texttt{pres}(P_d,t)$ denotes the number of preserved items by t, i.e., the items in the domain of the partial track morphisms of t, and $\Delta(g,d)$ denotes the maximum number of necessary deletions. Given an injective morphism $g\colon A \hookrightarrow G$ and a proper condition d, $\Delta(g,d)$ is defined inductively as follows: $\Delta(g,\texttt{true}) = 0$, $\Delta(g,\exists a) = 0$,

$$\begin{array}{lllll} (1) & \Delta(g,\nexists a) & = & \sum_{g' \in \text{Ext}(g)}(1 + dang(g')) \\ (2) & \Delta(g,\exists(a,c)) & = & \max_{g' \in \text{Ext}(g)} \Delta(g',c) \\ (3) & \Delta(g,\forall(a,c)) & = & \sum_{g' \in \text{Ext}(g)}(\Delta(g',c)) \end{array}$$

where $\text{Ext}(g) = \{g'\colon C \to G \mid g' \circ a = g\}$, and $dang(g')$ denotes the maximum number of dangling edges at g', i.e. $dang(g') = 0$ if there is some edge in $g'(C-A)$ and $\max_{v \in (C-A)} \text{inc}(v)$, otherwise, where, for a node v, $\text{inc}(v)$ denotes the number of edges incident to v.

Remark. Given a morphism $g\colon A \hookrightarrow G$ and a proper condition d, we determine the maximal number of necessary deletions $\Delta(g,d)$. This is zero if the condition is `true` or of the form $\exists a$. For a condition of the

[9] For a set S, $|S|$ denotes the number of elements.

form $\nexists a$, we consider all morphisms $g' \in \mathrm{Ext}(g)$, and sum up the number of deletions. For a condition of the form $\exists(a,c)$, we consider all morphisms $g' \in \mathrm{Ext}(g)$ and build the maximum of all $\Delta(g',c)$. For a condition of the form $\forall(a,c)$, we consider all morphisms $g' \in \mathrm{Ext}(g)$ and sum up the number of necessary deletions for the condition c at that position, i.e. $\Delta(g',c)$.

Fact 6. Repair program based on (1), in general, are neither terminating nor maximally preserving.

Proof. For the condition $\forall($  $,\exists$  $)$, $\langle$ $\Rightarrow$ $\rangle\downarrow$ is a repair program based on (1) not creating cycles. The same holds for $\langle$ $\Rightarrow$ $,\nexists$ $\rangle\downarrow$. Both programs are not terminating. A rule $C \Rightarrow A$ deletes $[C-A]$ items, although only one item has to be deleted, i.e., in general it is not maximally preserving. □

Lemma 3 (program properties). The repair program based on Construction 2(2) is terminating and maximally preserving.

Proof. The termination of the repair program based on (2) is shown in [8]. The maximal preservation of the repair program P_d based on (2) is shown by induction of the length of transformations: We show that, for all transformations $t\colon G \Rightarrow_{P_d,g,h} H$,

$$\mathtt{pres}(P_d,t) \geq \mathrm{size}(G) - \Delta(g,d).$$

Let d be a proper condition, P_d the repair program for d, and $t\colon G \Rightarrow_{P_d,g,h} H$ a transformation.

(1) For $d = \mathtt{true}$, $P_d = \mathtt{Skip}$, and $\mathtt{pres}(\mathtt{Skip},t) = \mathrm{size}(G)$.

(2) For $d = \exists a$, $P_d = \mathtt{try}\ \mathscr{R}_a$, and $\mathtt{pres}(\mathtt{try}\ \mathscr{R}_a,t) = \mathrm{size}(G)$.

(3) For $d = \nexists a$. $P_d = \mathscr{S}'_a\downarrow$. (a) If $g \models d$, then $\mathtt{pres}(\mathscr{S}'_a\downarrow,t) = \mathrm{size}(G)$. (b) If $g \not\models d$, then the transformation $G \Rightarrow^{+}_{\mathscr{S}'_a\downarrow,g,h} H$ is of the form $G \Rightarrow_{\mathscr{S}'_a,g,g_1} G_1 \Rightarrow_{\mathscr{S}'_a\downarrow,g_1,h} H$ where t_1 denotes the transformation starting with G_1. Then, for all $g' \in \mathrm{Ext}(g)$, (*) $\mathrm{size}(G_1) = \mathrm{size}(G) - \Delta(g',d)$ with $\Delta(g',d) = 1 + \mathit{dang}(g')$. By definition of Δ, (**) $\Delta(g,d) = \Delta(g',d) + \Delta(g_1,d)$.

$$\begin{array}{lcll} \mathtt{pres}(P_d,t) & \geq & \mathtt{pres}(P_d,t_1) & \\ & \geq & \mathrm{size}(G_1) - \Delta(g_1,d) & \text{(induction hypothesis)} \\ & = & \mathrm{size}(G) - \Delta(g,d) & \text{((*), (**))} \end{array}$$

(4) For $d = \exists(a,c)$, $P_d = P_{\exists a}; \langle \mathtt{Sel}(a); P_c; \mathtt{Uns}(a) \rangle$. (a) If $g \models d$, then $\mathtt{Pres}(P_d,G) = \mathrm{size}(G)$. (b) If $g \not\models d$, then $G \Rightarrow_{P_d,g,h} H$ is of the form $G \Rightarrow_{P_{\exists a},g,g'} G_1 \Rightarrow_{P'_c,g',h} H$. Then $\mathrm{size}(G_1) \geq \mathrm{size}(G)$ and, for all $g' \in \mathrm{Ext}(g)$, (**) $\Delta(g,d) = \Delta(g',c)$.

$$\begin{array}{lcll} \mathtt{pres}(P_d,t) & = & \mathtt{pres}(P'_c,t_1) & \\ & \geq & \mathrm{size}(G_1) - \Delta(g',c) & \text{(induction hypothesis)} \\ & = & \mathrm{size}(G) - \Delta(g,d) & \text{((*), (**))} \end{array}$$

(5) For $d = \forall(a,c)$, $P_d = \langle \mathtt{Sel}(a,\neg c); P_c; \mathtt{Uns}(a) \rangle\downarrow$. (a) If $g \models d$, then $\mathtt{Pres}(P_d,G) = \mathrm{size}(G)$. (b) If $g \not\models d$, then $G \Rightarrow_{P_d,g,h} H$ is of the form $G \Rightarrow_{P'_c,g,g_1} G_1 \Rightarrow_{P_d,g_1,h} H$ where P'_c denotes the program without iteration. If c is of the form $\exists(a',c')$, then, $\mathrm{size}(G_1) \geq \mathrm{size}(G)$ and, for every $g' \in \mathrm{Ext}(g)$, $\Delta(g',c) = 0$. If c is of the form $\nexists a'$, then, for every $g' \in \mathrm{Ext}(g)$, $\mathrm{size}(G_1) = \mathrm{size}(G) - \Delta(g',c)$ as in Case (3). Thus, (*) $\mathrm{size}(G_1) \geq \mathrm{size}(G) - \Delta(g',c)$. By definition of Δ, (**) $\Delta(g,d) = \Delta(g_1,d) + \Delta(g',c)$.

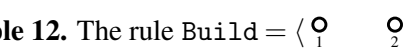
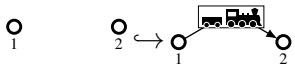
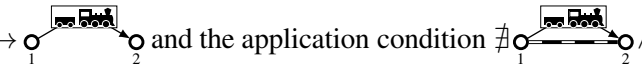
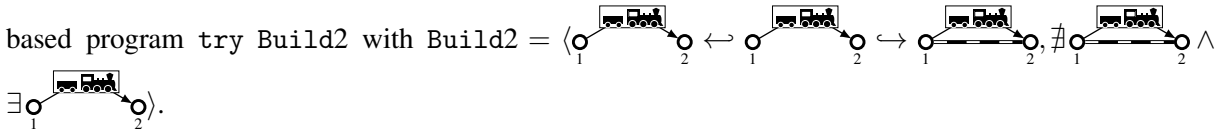
$$\begin{array}{lcll} \texttt{pres}(P_d,t) & \geq & \texttt{pres}(P_d,t_1) & \\ & \geq & \text{size}(G_1)-\Delta(g_1,d) & \text{(induction hypothesis)} \\ & \geq & \text{size}(G)-\Delta(g,d) & \text{((*), (**))} \end{array}$$

This completes the inductive proof. □

4 Rule-based repair

A rule-based program is a program based on a set of rules equipped with the dangling-edges operator, context, application condition, and interface.

Definition 11 (rule-based programs). Given a set of rules $\mathscr{R}$, a program is *$\mathscr{R}$-based*, if all rules in the program are rules in $\mathscr{R}$ equipped with dangling-edges operator, context, application condition, and interface. Additionally, the empty program `Skip` is $\mathscr{R}$-based.

Example 12. The rule $\texttt{Build} = \langle$  $\rangle$ equipped with the context  and the application condition  yields to the $\{\texttt{Build}\}$-based program `try Build2` with $\texttt{Build2} = \langle$  $\rangle$.

The construction of an $\mathscr{R}$-based repair program for a proper condition d is based on the following idea (see Figure 5).

(1) Take a repair program for the condition d (Theorem 1).

(2) Try to refine the rules of the repairing sets by equivalent transformations via $\mathscr{R}$.

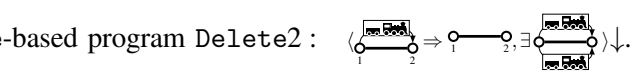
(3) Transform the transformations into equivalent $\mathscr{R}$-based programs (Theorem 2).

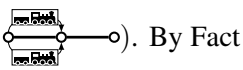
(4) Replace each repairing set in P_d by the equivalent $\mathscr{R}$-based program (Theorem 3).

In the following, we introduce the main notion of compatibility, saying that, for all rules of the repairing sets of the repair program for d, there are equivalent transformations via the rule set.

Definition 12 (equivalence). Two transformations t,t' from G to H are *equivalent*, denoted $t \equiv t'$, if for each extension form G^* to H^* there is an extension of t' from G^* to H^*, and vice versa.

Definition 13 (compatibility). Let d be a proper condition. A set of rules $\mathscr{R}$ is *d-compatible* (w.r.t. a repair program P_d) if, for all rules in the repairing sets of P_d, there are *e*quivalent transformations via $\mathscr{R}$. In particular, if $\mathscr{R} = \{\rho\}$, we also say that ρ is d-compatible.

Example 13. Let NoTwo $= \nexists(\emptyset \hookrightarrow$  $)$. Then $\{\texttt{Delete}\}$ is a repairing set for NoTwo, and there is a transformation  via the `Delete`-based program `Delete2` :  .

The rule set $\{\texttt{Move}\}$ is a repairing set for $\nexists(\emptyset \hookrightarrow$  $)$. By Fact 5, the rule set $\{\texttt{Move},\texttt{Delete}\}$ is a repairing set for NoTwo.

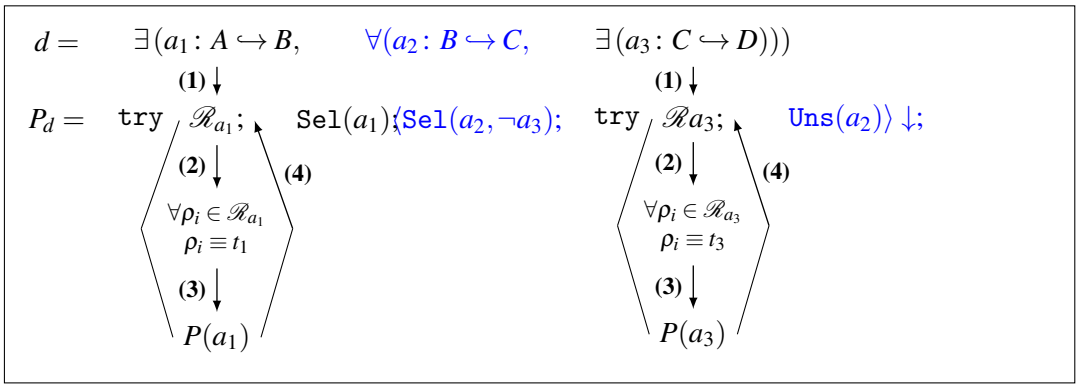


Figure 5: Construction of an $\mathscr{R}$-based repair program

Example 14 (no {Move,Delete}-based repair). Consider the constraint Station $= \exists$  (there exists a train station). Whenever the start graph has no station, then no station can be created by a {Move,Delete}-based program. The reason is that the labels of the constraint do not occur in the right-hand sides of the rules.

In the case of $\mathscr{R}$ is d-compatible w.r.t. P_d, for all rules in the repair program, there are transformations via $\mathscr{R}$. These transformations via $\mathscr{R}$ can be transformed into $\mathscr{R}$-based programs.

Theorem 2 (from transformations to rule-based programs). For every transformation $t\colon G \Rightarrow^*_{\mathscr{R}} H$, there is a $\mathscr{R}$-based program $P(t)$ such that $t \equiv P(t)$.

Construction 4. Let $t : G \Rightarrow^*_{\mathscr{R}} H$ be a transformation. For direct transformations $t\colon G \Rightarrow_{\rho,g,h} H$ via a rule $\rho = \langle x, p, \mathrm{ac}, y\rangle$ with interfaces X and Y, let $P(t) := \langle \mathtt{Sel}(g' \circ x, \mathrm{ac}'); \overline{\rho}'; \mathtt{Uns}(h' \circ y)\rangle$ be the rule with left interface $g' \circ x$, $\overline{\rho} = G \Rightarrow H$ be the rule ρ equipped with context, $\mathrm{ac}' = \mathrm{Shift}(g', \mathrm{ac})$ the left application condition for $\overline{\rho}$, and $h' \circ y$ the right interface. For transformations $t\colon G = G_0 \Rightarrow^{n+1}_{\mathscr{R}} G_{n+1} = H$, with $t_1\colon G_0 \Rightarrow^n_{\mathscr{R}} G_n$, and $t_2\colon G_n \Rightarrow_\rho G_{n+1}$, $P(t) := \langle P(t_1); P(t_2)\rangle$.

Proof. Let $t : G \Rightarrow^*_{\mathscr{R}} H$ be a transformation. By construction, $P(t)$ is $\mathscr{R}$-based. We show that there is a transformation $G \Rightarrow_{P(t)} H$. For one-step transformations, by construction, $t \equiv P(t)$. For $n+1$-step transformations, by induction hypothesis, $t_1 \equiv P(t_1)$ and $t_2 \equiv P(t_2)$. Then $P(t) := \langle P(t_1); P(t_2)\rangle$ is a program with $t \equiv P(t)$. □

Theorem 3 (rule-based repair). For every proper condition d, every repair program P_d for d, and every rule set $\mathscr{R}$ d-compatible w.r.t. P_d, there is an $\mathscr{R}$-based repair program for d.

Construction 5. Let $P'_d = P_d[\mathrm{repl}]$ where repl is constructed as follows: By assumption, for the rules in the repairing A-set, there are equivalent transformations via $\mathscr{R}$. For these transformations, there are equivalent $\mathscr{R}$-based programs with interfaces (Theorem 2). The mapping repl replaces the repairing A-sets by equivalent $\mathscr{R}$-based programs with interfaces A.

Proof. By assumption, for all rules in the repairing A-sets of P_d, there are equivalent transformations via $\mathscr{R}$. By Theorem 2, the transformations can be transformed into equivalent $\mathscr{R}$-based repair programs.

This yields a mapping repl which replaces the repairing A-sets by equivalent $\mathscr{R}$-based programs with interfaces A. By the Leibniz's replacement principle, the repair programs P_d and $P_d[\text{repl}]$ are equivalent. Thus, $P_d[\text{repl}]$ is an $\mathscr{R}$-based repair program for d. □

To get maximally preserving rule-based repair programs, we have to assume, that our input rule set is maximally preserving, as well. If a rule set is maximally preserving, then the number of deleted items is minimal. For non-deleting rule sets, the number of deleted elements is 0, and the graph can be preserved. If a rule set is deleting, we delete edges instead of nodes, whenever possible, since it is more costly to delete nodes than edges.

Fact 7 (program properties). Rule-based repair programs based on Construction 2(1), in general, are neither terminating nor maximally preserving. Rule-based repair programs based on Construction 2(2) are terminating and maximally preserving.

Proof. The statements follow immediately form the corresponding statements in Fact 3. □

5 Related concepts

In this section, we present some related concepts on rule-based graph repair. For the related problem of model repair, there is a wide variety of different approaches. For a more sophisticated survey on different model repair techniques, and a feature-based classification of these approaches, see **Macedo et al.** [10].

Rule-based repair. The notion rule-based repair is used in different meanings. In most cases [11, 8], a rule set is derived from a set of constraints and a repair algorithm/program is constructed from the rule set. In this paper, a rule set and a condition are given as input and a repair program is constructed from the rule set.

In **Nassar et al. 2017** [11], 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 consider the EMF model constraints defined in [2].

In **Habel and Sandmann 2018** [8], given a proper condition, we derive a rule set from the condition d and construct a repair program using this rule set. The repair program is required to repair all graphs. In this paper, we use programs with interface [14] with selection and unselection of parts, instead of markings as in [8]. For simple cases and illustration purposes, marking may be an alternative. For conditions with large nesting depth, the morphism-based concept is more convenient, the marking of elements requires an additional marking for each nesting depth.

In **Schneider et al. 2019** [17], 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.

All approaches are proven to be **correct**, i.e. the repair (programs) yield to a graph satisfying the condition. In Schneider et al. [17] 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 ($\mathscr{R}$-based) repair program, then apply this program to an arbitrary graph. In Schneider et al. [17],

	Schneider et al. [17]	Nassar et al. [11]	Habel et al. [8]	this work
input	FO condition & graph	EMF model	FO condition	FO condition & rule set
output	repair algorithm	repair rules & valid model	repair program	repair program
correctness	+	+	+	+
dynamic	+	-	-	-
termination	-	(+)	+	(+)

Table 1: Overview of selected repair approaches

the program does not **terminate**, if the repair updates trigger each other ad infinitum. If we choose the repairing set accordingly, we get a terminating repair program.

In **Taentzer et al. 2017** [19], a designer can specify a set of so-called change-preserving rules, and a set of edit rules. Each edit rule, which yields 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 **Cheng et al. 2018** [3], a rule-based approach for graph repair is presented. Given a set of rules, and a graph, they use this set of rules, to handle different kinds of conditions, i.e., incompleteness, conflicts and redundancies. The rules are based on seven different operations not defined in the framework of the DPO-approach. They look for the "best" repair based on the "graph edit distance".

6 Conclusion

The repair programs are formed from rules derived from the given proper condition. They were constructed to be maximally preserving, i.e. to preserve nodes as much as possible. Additionally, we have considered rule-based repair where the repair programs are constructed from a given set of small rules and a given condition. Based on the repair program for proper conditions, we have constructed a rule-based repair program for proper conditions provided that the given rule set is compatible with the repairing sets of the original program.

Summarizing, we have constructed

(1) repair programs for proper conditions (Theorem 1),

(2) rule-based programs from transformations (Theorem 2),

(3) rule-based repair programs for proper conditions provided that the given rule set is compatible with the repairing sets of the original program (Theorem 3).

Further topics are rule-based repair programs for all satisfiable conditions, for typed attributed graphs and EMF-models, i.e., typed, attributed graphs satisfying some constraints, and an implementation. The aim is to represent the structure of a meta model as graph-like structure, and OCL constraints as nested graph conditions, and then use the ($\mathscr{R}$-based) graph repair for ($\mathscr{R}$-based) model repair.

Acknowledgements. We are grateful to Marius Hubatschek, Jens Kosiol, Nebras Nassar, and the anonymous reviewers for their helpful comments to this paper.

References

[1] Gábor Bergmann (2014): *Translating OCL to Graph Patterns*. In: *Model-Driven Engineering Languages and Systems (MODELS 2014)*, LNCS, pp. 670–686, doi:10.1007/978-3-319-11653-2_41.

[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] Yurong Cheng, Lei Chen, Ye Yuan & Guoren Wang (2018): *Rule-Based Graph Repairing: Semantic and Efficient Repairing Methods*. In: *34th IEEE International Conference on Data Engineering, ICDE 2018,*, pp. 773–784, doi:10.1109/ICDE.2018.00075.

[4] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science, Springer.

[5] Hartmut Ehrig, Claudia Ermel, Ulrike Golas & Frank Hermann (2015): *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science, Springer, doi:10.1007/978-3-662-47980-3.

[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/S0960129500001353.

[7] Annegret Habel & Detlef Plump (2001): *Computational Completeness of Programming Languages Based on Graph Transformation*. In: *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, *Lecture Notes in Computer Science* 2030, pp. 230–245, doi:10.1007/BFb0017401.

[8] 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/s10009-018-0496-3.

[9] 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.

[10] 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.

[11] 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:10.1007/978-3-319-21145-9_10.

[12] 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.

[13] OMG: *Object Constraint Language*. `https://www.omg.org/spec/OCL/`.

[14] Karl-Heinz Pennemann (2009): *Development of Correct Graph Transformation Systems*. Ph.D. thesis, Universität Oldenburg.

[15] Detlef Plump (2005): *Confluence of Graph Transformation Revisited*. In: *Processes, Terms and Cycles: Steps on the Road to Infinity*, *Lecture Notes in Computer Science* 3838, pp. 280–308, doi:10.1007/BF00289616.

[16] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel & Grabriele Taentzer (2018): *Translating Essential OCL Invariants to Nested Graph Constraints for Generating nstances of Meta-models*. *Science of Computer Programming* 152, pp. 38–62, doi:10.1016/j.scico.2017.08.006.

[17] 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:10.1007/978-3-662-54494-5_16.

[18] Andy Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In: *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, pp. 151–163. Available at `https://doi.org/10.1007/3-540-59071-4_45`.

[19] 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:10.1007/11880240_15.