Graph repair by graph programs[⋆]

Annegret Habel, Christian Sandmann

Universität Oldenburg
{habel,sandmann}@informatik.uni-oldenburg.de

Abstract. Model repair is an essential topic in meta-modelling. We consider the problem of graph repair: Given a graph constraint d, we try to construct a graph program, such that the application to any graph G yields a graph H satisfying the graph constraint d. We show the existence of terminating repair programs for a number of satisfiable constraints of nesting depth ≤ 2.

1 Introduction

In model-driven software engineering the primary artifacts are meta-models, which have to be consistent wrt. a set of constraints (see e.g. [EEGH15]). 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. [NEF03,MGC13,NKR17]). Since meta-models can be represented as graph-like structures [BET12], in this paper, we consider the problem of *graph repair*: Given a graph constraint d, we try to construct a graph program, called *repair program*, such that the application to any graph G yields a graph H satisfying the graph constraint d.

Repair problem

Given: A graph constraint d.
Task: Find a graph program P: $\forall G \Rightarrow_P H,\ H \models d$.

$$\begin{tikzcd} {} \arrow[r, "\text{constraint } d"] & \boxed{\begin{array}{c}\text{program}\\ \text{construction}\end{array}} \arrow[r, "\text{program } P", "\forall G \Rightarrow_P H. H \models d"'] & {} \end{tikzcd}$$

More specifically, we look for repair programs that are terminating and maximally preserving, meaning that an input graph is preserved as long as there is no requirement for non-existence of items.

In this paper, we focus on constraints of nesting depth ≤ 2, in particular of the form $\forall(A, \exists\, C)$ meaning that, for all occurrences of the graph A, there exists an occurrence of the supergraph C, and $\exists\,(A, \nexists\, C)$, meaning that there exists

⋆ 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).

an occurrence of the graph A, but, at that place, not an occurrence of the supergraph C. We show that there are terminating repair programs for

1. constraints of the form $\forall(A, \exists\, C)$ and $\exists\,(A, \nexists\, C)$,
2. for the conjunction $d_1 \wedge d_2$ provided that there are terminating repair programs P_i for the subconstraints d_i $(i = 1, 2)$ and P_1 preserves the constraint d_2 or P_2 preserves the constraint d_1, and
3. for the disjunction $d_1 \vee d_2$ provided that there are terminating repair programs P_i for some subconstraint d_i $(i \in \{1, 2\})$.

We illustrate our approach with Petri nets as the modeling language. We express properties of Petri nets such as "every place has at most/at least one token", "the number of places is less or equal k" by graph conditions. The example is a simplification of the one in [RAB$^+$18] for typed attributed graphs.

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, the main section of the paper, we introduce repair programs and present some constructions of repair programs for satisfiable constraints of nesting depth ≤ 2. In Section 4, we present some related concepts. In Section 5, 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 graph programs [EEPT06,HP09].

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 $\mathcal{L}$) 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 $\mathrm{l}_{\mathrm{V},G}\colon \mathrm{V}_G \to \mathcal{L}$, $\mathrm{l}_{\mathrm{E},G}\colon \mathrm{E}_G \to \mathcal{L}$ are 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, $g_\mathrm{V} \circ \mathrm{s}_G = \mathrm{s}_H \circ g_\mathrm{E}$, $g_\mathrm{V} \circ \mathrm{t}_G = \mathrm{t}_H \circ g_\mathrm{E}$, $\mathrm{l}_{\mathrm{V},G} = \mathrm{l}_{\mathrm{V},H} \circ g_\mathrm{V}$, $\mathrm{l}_{\mathrm{E},G} = \mathrm{l}_{\mathrm{E},H} \circ g_\mathrm{E}$. 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 $g_\mathrm{V}(v) = v$ and $g_\mathrm{E}(e) = e$ for all $v \in \mathrm{V}_G$ and all $e \in \mathrm{E}_G$.

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 [HP09].

Definition 2 (graph conditions). A *(graph) condition* over a graph P is of the form (a) $\mathtt{true}$ or (b) $\exists(a, c)$ where $a\colon P \hookrightarrow C$ is an inclusion morphism and c is a condition over C. For conditions c, c_i ($i \in I$ for some finite index set I[1]) over P, $\neg c$ and $\wedge_{i\in I} c_i$ are conditions over P. Conditions over the empty graph $\emptyset$ are called *constraints*. In the context of rules, conditions are called *application conditions.*

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

Definition 3 (semantics). Any injective morphism $p\colon P \hookrightarrow G$ *satisfies* $\mathtt{true}$.

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

$$\begin{tikzcd}
\exists\,(\ P \arrow[rr, hook, "a"] \arrow[dr, hook, "p"'] & {=} & C, \arrow[dl, hook', "q"] & \triangleleft\ c\ ) \\
& G & & \not\models
\end{tikzcd}$$

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 P). A graph G *satisfies* a constraint c, $G \models c$, if the morphism $p\colon \emptyset \hookrightarrow G$ satisfies c. $[\![c]\!]$ denotes the class of all graphs satisfying c. A constraint c is *satisfiable* if there exists a graph G such that $G \models c$. A constraint c is *valid* if all graphs satisfy c.
Two conditions c and c' over P are *equivalent*, denoted by $c \equiv c'$, if for all graphs G and all injective morphisms $p\colon P \hookrightarrow G$, $p \models c$ iff $p \models c'$. A condition c *implies* a condition c', denoted by $c \Rightarrow c'$, if for all graphs and all injective morphisms $p\colon P \hookrightarrow G$, $p \models c$ implies $p \models c'$.

Constraints can be "shifted" over morphisms into application conditions.

Lemma 1 (Shift **[HP09]).** There is a Shift-construction such that, for each condition d over P and for each injective morphism $b\colon P \hookrightarrow R$, Shift transforms d via b into a condition $\mathrm{Shift}(b, d)$ over R such that, for each injective morphism $n\colon P \hookrightarrow H$, $n \circ b \models d \Leftrightarrow n \models \mathrm{Shift}(b, d)$.

Construction. For a condition d over P and an injective morphism $b\colon P \hookrightarrow R$, the condition $\mathrm{Shift}(b, d)$ over R is defined as follows.

$$\begin{tikzcd}
P \arrow[r, hook, "b"] \arrow[d, hook, "a"'] \arrow[dr, phantom, "(0)"] & R \arrow[d, hook, dashed, "a'"] \\
C \arrow[r, hook, dashed, "b'"'] & R' \\
\underset{c}{\triangle} & \triangle
\end{tikzcd}$$

$\mathrm{Shift}(b, \mathtt{true}) := \mathtt{true}$.
$\mathrm{Shift}(b, \exists\ (a, d)) := \bigvee_{(a',b')\in\mathcal{F}} \exists\ (a', \mathrm{Shift}(b', d))$ where
$\mathcal{F} = \{(a', b') \mid b' \circ a = a' \circ b,\ a', b' \text{ inj},\ (a', b') \text{ jointly surjective}^2\}$
$\mathrm{Shift}(b, \neg d) := \neg\mathrm{Shift}(b, d)$, $\mathrm{Shift}(b, \wedge_{i\in I} d_i) := \wedge_{i\in I}\mathrm{Shift}(b, d_i)$.

[1] In this paper, we consider graph conditions with finite index sets.
[2] 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 $b'(z) = x$.

Rules are specified by a pair of inclusion morphisms. For restricting the applicability of rules, the rules are equipped with a left application condition. Such a rule is applicable with respect to an injective "match" morphism from the left-hand side of the rule to a graph, if, and only if, the underlying plain rule is applicable and the match morphism satisfies the left application condition.

Definition 4 (rules and transformations). A *rule* $\varrho = \langle p, \mathrm{ac} \rangle$ consists of a *plain rule* $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ with inclusion morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$ and an application condition ac over L. It is *non-deleting* if $L \cong K$ and *deleting* if $L \supset K$. A rule $\langle p, \mathtt{true} \rangle$ is abbreviated by p. A *direct transformation* from a graph G to a graph H applying rule ϱ requires the following steps:

(1) Find an injective morphism $g \colon L \hookrightarrow G$ such that $g \models \mathrm{ac}$.
(2) Delete all edges in $G - g(L)$ incident to a node in $g(L - K)$ yielding G'[3].
(3) Apply the rule ϱ to G' at the restricted morphism $g' \colon L \hookrightarrow G'$,
i.e., delete $g(L - K)$ from G', yielding D and add $R - K$ to D, yielding H.

$$\begin{tikzcd}
\mathrm{ac} \arrow[r, phantom, "\triangleright"] & L \arrow[dl, "g"'] \arrow[d, hook, "g'"] \arrow[dl, phantom, "=" near end] \arrow[dr, phantom, "(1)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook, "d"'] \arrow[dr, phantom, "(2)"] & R \arrow[d, hook, "h"] \\
G & G' \arrow[l, hook', "\mathrm{inc}"] & D \arrow[l, hook', "\mathrm{inc}_{G'}"] \arrow[r, hook, "\mathrm{inc}_H"'] & H
\end{tikzcd}$$

We write $G \Rightarrow_{\varrho,g} H$ or $G \Rightarrow_{\varrho} H$ if there exists such a direct transformation. A *transformation* is a sequence of direct transformations. A transformation from G to H by a rule set $\mathcal{R}$ is denoted by $G \Rightarrow^*_{\mathcal{R}} H$ or $G \Rightarrow^* H$.

Notation. A rule $\langle L \hookleftarrow K \hookrightarrow R \rangle$ sometimes is denoted by $L \Rightarrow R$ where indexes in L and R refer to the corresponding nodes.

Remark. In the following, we use special injective rules and injective matches. For node-deleting rules, we have to allow the deletion of dangling edges, but we do not use the full power of the SPO-approach. Thus, we speak about the DPO-approach with deletion of dangling edges.

With every transformation $t \colon G \Rightarrow^* H$ a partial 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). The *track morphism* $\mathrm{track}_{G \Rightarrow H}$ from G to H is the partial morphism[4] defined by $\mathrm{track}_{G \Rightarrow H}(x) = \mathrm{inc}_H(\mathrm{inc}_G^{-1}(x))$ if $x \in D$ and *undefined* otherwise, where $\mathrm{inc}_G = \mathrm{inc} \circ \mathrm{inc}_{G'}$ and $\mathrm{inc}_G^{-1} \colon \mathrm{inc}_G(D) \hookrightarrow D$

[3] These edges are said to be *dangling edges.*
[4] A *partial* graph morphism $G \rightharpoonup H$ is an inclusion $S \hookrightarrow H$ such that $S \subseteq G$.

is the inverse of inc_G. Given a transformation $G \Rightarrow^* H$, $\mathrm{track}_{G\Rightarrow^* H}$ is defined by induction on the length of the transformation: $\mathrm{track}_{G\Rightarrow^* H} = \mathrm{iso}$ for an isomorphism $\mathrm{iso}\colon G \overset{\sim}{\to} H$, and $\mathrm{track}_{G\Rightarrow^* H} = \mathrm{track}_{G'\Rightarrow H} \circ \mathrm{track}_{G\Rightarrow^* G'}$ for $G \Rightarrow^* H = G \Rightarrow^* G' \Rightarrow H$.

Graph programs are made of rules with application conditions closed under non-deterministic choice, sequential composition, and iteration.

Definition 6 (graph programs). *(Graph) programs* are inductively defined:

(1) Every rule is a program.
(2) Every finite set of programs is a program.
(3) Given programs P and Q, then $\langle P; Q\rangle$ and $P\downarrow$ are programs.

The *semantics* of a program P is a binary relation $[\![P]\!]$ on graphs:

(1) For every rule ϱ, $[\![\varrho]\!] = \{\langle G, H\rangle \mid G \Rightarrow_\varrho H\}$.
(2) For a finite set $\mathcal{S}$ of programs, $[\![\mathcal{S}]\!] = \cup_{P\in\mathcal{S}}[\![P]\!]$[5].
(3) For programs P and Q, $[\![(P;Q)]\!] = [\![Q]\!] \circ [\![P]\!]$ and $[\![P\downarrow]\!] = \{\langle G, H\rangle \in [\![P]\!]^* \mid \neg\exists M.\ \langle H, M\rangle \in [\![P]\!]\}$ where $[\![P]\!]^*$ is the reflexive, transitive closure of $[\![P]\!]$.

For $\langle G, H\rangle \in [\![P]\!]$, we also write $G \Rightarrow_P H$. A program P is *terminating* if there is no infinite transformation. For a finite rule set $\mathcal{R}$, `Try` $\mathcal{R}$ denotes the one-fold application of $\mathcal{R}$, if $\mathcal{R}$ is applicable, and the zero-fold application, otherwise.

Programs according to (1) are *elementary*, (2) are *choice programs*, the program $\langle P; Q\rangle$ is the *sequential composition* of P and Q, and $P\downarrow$ is the *iteration* of P. Programs of the form $\langle P; \langle Q; R\rangle\rangle$ and $\langle\langle P; Q\rangle; R\rangle$ are considered as equal; by convention, both can be written as $\langle P; Q; R\rangle$.

3 From graph constraints to repair programs

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

A repair program for a constraint is a graph program with the property that any application to a graph yields a graph satisfying the constraint. For satisfiable constraints, it is easy to create such a repair program, e.g., by deleting the input and creating a graph satisfying the constraint.

We define maximally preserving repair programs where items are preserved whenever possible. Whenever a graph satisfies a constraint, but the negation is required, then the graph cannot be repaired without deletions. In this case, at least as necessary items should be deleted.

[5] In particular, for the empty set S, $[\![S]\!] = \emptyset$.

Definition 7. For a graph G and a constraint d,

$$\Delta(G,d) = \begin{Bmatrix} \min_{p \in \mathrm{Mor}(A,G)} |\mathrm{Ext}(p)|^6 & \text{if } d = \exists\,(A, \nexists\, C) \\ 0 & \text{otherwise} \end{Bmatrix}$$

where $\mathrm{Mor}(A,G)$ denotes the set of all injective morphisms from A to G and $\mathrm{Ext}(p) = \{q\colon C \to G \mid q|_A = p\}$ denotes the set of extensions of $p\colon A \hookrightarrow G$.

Given a transformation t, $\mathrm{pres}(t)$ denotes the number of items in the domain of the track morphism track_t. Let $\mathcal{T}$ denote the set of transformations $t\colon G \Rightarrow_P H$ starting at a graph G. For a program P and a graph G, $\mathrm{Pres}(P,G)$ denotes the maximum of $\mathrm{pres}(t)$, i.e., $\mathrm{Pres}(P,G) = \max_{t \in \mathcal{T}} \mathrm{pres}(t)$.

Definition 8 (repair programs). A *repair program* for a graph constraint c is a graph program P such that, for all transformations $G \Rightarrow_P H$, $H \models c$. It is *maximally preserving* if for each graph G, $\mathrm{Pres}(P,G) \geq \mathrm{size}(G) - \Delta(G,d)$.

Remark. A repair program P for d is *destructive*, if it is of the form $P = \langle \mathtt{Delete}; P_\emptyset \rangle$ where `Delete` deletes the input graph and $P_\emptyset$ applied to the empty graph yields a graph satisfying d. For destructive programs P, non-empty graphs G, and constraints $d = \forall(A, \exists\, C)$, $\mathrm{Pres}(P,G) = 0$, thus, destructive programs are not maximally preserving.

For nested graph conditions, the nesting depth may be defined as follows.

Definition 9 (nesting depth). The conditions `true` and `false` are of nesting depth 0. If $a\colon P \hookrightarrow C$ and c is a condition over C of nesting depth $i \geq 0$, then the conditions $\exists\,(a,c), \nexists\,(a,c)$ are of nesting depth $i+1$.

In the following subsections, we consider constraints of nesting depth ≤ 2. Most important are the "atomic" constraints $\forall(A, \exists\, C)$ and $\exists\,(A, \nexists\, C)$ without conjunctions and disjunctions. All other atomic constraints of nesting depth ≤ 2 can be expressed with the help of these constraints (see Table 1).

3.1 Constraints of the form $\forall(A, \exists\, C)$

In this subsection, we consider constraints of the form $\forall(A, \exists\, C)$, with $A \ncong C$, informally meaning that, for all occurrences of the graph A, there exists an occurrence of the supergraph C. For these constraints, $\langle A \Rightarrow C, \nexists\, C\rangle\downarrow$ seems to be an easy and intuitive repair program, but it need not be terminating: By $A \subset C$, the occurrence of the rule remains preserved, the rule can be applied infinitely often, and there is an infinite transformation $A \Rightarrow C \Rightarrow C_1 \Rightarrow C_2 \ldots$ with $C_i \subset C_{i+1}$ for $i \geq 1$.

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

Example 1. For $\forall(\underset{1}{\bullet}, \exists\ \underset{1}{\bullet}\!\rightarrow\!\bullet)$ meaning that, for every node, there has to exist a proper outgoing edge, $\langle \underset{1}{\bullet} \Rightarrow \underset{1}{\bullet}\!\rightarrow\!\bullet, \nexists\ \underset{1}{\bullet}\!\rightarrow\!\bullet \rangle\downarrow$ is a repair program not creating cycles. Unfortunately, the program is not terminating.

Nevertheless, we use this program as a first step to construct a terminating one: The program with rules of the form $B \Rightarrow C$ where $A \subseteq B \subset C$ and an application condition requiring that no larger subgraph B' of C ($\wedge_{B'} \nexists B'$) occurs and the shifted condition $\nexists\,(A \hookrightarrow C)$ is satisfied, is a terminating repair program.

Theorem 1. For constraints of the form $\forall(A, \exists\, C)$, with $A \ncong C$, a terminating, maximally preserving repair program can be effectively constructed.

Construction 1. For $\forall(A, \exists\, C)$, with $A \ncong C$, let $\mathcal{R}\!\downarrow$ with

$$\mathcal{R} = \{\langle B \Rightarrow C, \mathrm{ac}_B \wedge \mathrm{ac}\rangle \mid A \subseteq B \subset C\}$$

where $\mathrm{ac}_B = \bigwedge_{B'} \nexists\, B'$, $\bigwedge_{B'}$ ranges over all $\nexists\, B'$ with $B \subset B' \subseteq C$ and $\mathrm{ac} = \mathrm{Shift}(A \hookrightarrow B, \nexists\,(A \hookrightarrow C))$.

Remark. The rule set $\mathcal{R}$ is — up to isomorphism — a finite, non-empty set of non-deleting rules. By the application condition, each rule can only be applied if the constraint is not satisfied and no other rule whose left-hand side includes B and is larger can be applied. By $B \subset C$, the rule set $\mathcal{R}$ does not contain identical rules.

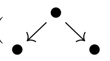
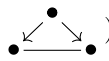
Example 2. For the constraint alltok $= \forall(\boxed{\text{Pl}}, \exists\,(\boxed{\text{Pl}}\xrightarrow{\text{token}}\boxed{\text{tk}}))$, meaning each place has a token, we obtain the program $\mathcal{R}_1\downarrow$, with [7]

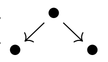
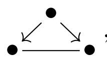
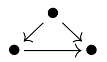
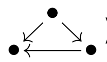
$$\mathcal{R}_1 = \left\{ \begin{array}{l} \langle \boxed{\text{Pl}} \quad\quad \Rightarrow \boxed{\text{Pl}}\xrightarrow{\text{token}}\boxed{\text{tk}}, \nexists\,(\boxed{\text{Pl}}\ \boxed{\text{tk}})\rangle \\ \langle \boxed{\text{Pl}}\ \boxed{\text{tk}} \Rightarrow \boxed{\text{Pl}}\xrightarrow{\text{token}}\boxed{\text{tk}}, \nexists\,(\boxed{\text{Pl}}\xrightarrow{\text{token}}\boxed{\text{tk}}) \wedge \nexists\,(\boxed{\text{tk}}\ \boxed{\text{Pl}}\xrightarrow{\text{token}}\boxed{\text{tk}})\rangle \end{array} \right\}$$

This program is terminating: Given a place, the first rule adds a token and an edge to the token, if there does not exist a place and a token. The second rule requires the existence of a place and a token, and, if there is no edge from the place to the token and no edge to another token, then a connecting edge is added. The first application condition is used for termination: it cannot add an edge from a place to a token if there is already a connecting edge. The second application condition forbids the addition of an edge from the place to the token if there is already a connection to another token.

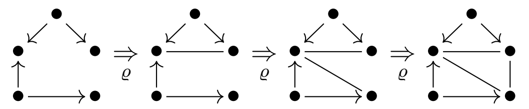
The following example shows that it may be not enough to repair all violations in a given graph, because new violations may occur when applying repair rules. Thus, termination is not trivial.

[7] The match of the rule is marked in a blue color.

Example 3. Let $\forall($  $, \exists$  $)$[8] . Then, the repair program $\mathcal{R}{\downarrow}$ contains the rule

$\varrho = \langle$  $\Rightarrow$  $, \nexists$  $\wedge \nexists$  $\rangle.$

Applying the rule to the graph below, we obtain the following transformation:

$G =$ 

There are two matches for the rule ϱ in the graph G. Applying the rule ϱ twice, yields a new match for ϱ, not already in G.

Proof (of Theorem 1). Let $\mathcal{R}{\downarrow}$ be the program according to Construction 1. For simplicity, we do the proof for unlabelled graphs. The proof for labelled graphs is similar.

$\mathcal{R}{\downarrow}$ is terminating: Let $G \Rightarrow H$ be a direct transformation through a rule $\varrho = \langle B \Rightarrow C, \mathrm{ac}_B \wedge \mathrm{ac}\rangle$ in $\mathcal{R}$. Then $|\mathrm{V}_B[\leq |\mathrm{V}_G|$. (1) If $|\mathrm{V}_G| < |\mathrm{V}_C|$ then $|\mathrm{V}_B| = |\mathrm{V}_G|$ and, by the application condition ac_B (there is no larger B' with $B \subset B' \subseteq C$), $|\mathrm{V}_C| = |\mathrm{V}_H|$. (2) If $|\mathrm{V}_G| \geq |\mathrm{V}_C|$ and (a) $\mathrm{E}_C = \emptyset$, then $|\mathrm{E}_B| = |\mathrm{E}_C|$ and $|\mathrm{E}_G| = |\mathrm{E}_H|$ and, by the application condition ac_B, the program is terminating. (b) If $\mathrm{E}_C \neq \emptyset$, then $|\mathrm{V}_B| = |\mathrm{V}_C|$ and $|\mathrm{V}_G| = |\mathrm{V}_H|$. Let k denote the maximal number of parallel edges in C, by the application condition ac_B, $\mathcal{R}{\downarrow}$ adds maximal k parallel edges in H, thus, $|\mathrm{E}_H| \leq k \cdot |\mathrm{V}_H| \times |\mathrm{V}_H|$.

$\mathcal{R}{\downarrow}$ is correct: Let $G \Rightarrow_{\mathcal{R}\downarrow} H$. By the semantics of $\downarrow$, no rule of $\mathcal{R}$ is applicable to H. Thus, for every rule $\langle B \Rightarrow C, \mathrm{ac}_B \wedge \mathrm{ac}\rangle$ in $\mathcal{R}$ and every injective morphism $h\colon B \hookrightarrow H$, the application condition $\mathrm{ac} = \mathrm{Shift}(A \hookrightarrow B, \nexists C)$ with $a\colon A \hookrightarrow C$, $b\colon A \hookrightarrow B$ is violated, i.e., $\mathrm{Shift}(A \hookrightarrow B, \exists C)$ is satisfied. By the Shift Lemma, for every morphism $p = h \circ b\colon A \to H$, the application condition $\exists C$ is satisfied. Consequently, H satisfies the constraint $\forall(A, \exists C)$.

$$\begin{aligned} &\forall h\colon B \hookrightarrow H.h \not\models \mathrm{Shift}(b, \nexists C)\\ \Leftrightarrow\ &\forall h\colon B \hookrightarrow H.h \models \mathrm{Shift}(b, \exists C)\\ \Leftrightarrow\ &\forall p\colon A \hookrightarrow H.p \models \exists C\\ \Leftrightarrow\ &H \models \forall(A, \exists C)\end{aligned}$$

$$\begin{tikzcd} & A \arrow[dl, hook, "b"'] \arrow[dr, hook, "a"] \arrow[dl, bend right=60, "p"', to path={-- (\tikztotarget)}] & \\ B \arrow[rr, hook, "c"'] \arrow[d, hook, "h"] & = & C \\ H & & \end{tikzcd}$$

$\mathcal{R}{\downarrow}$ is maximally preserving: By construction, the rules in $\mathcal{R}$ are non-deleting, i.e. for each graph G, $\Delta(\mathcal{R}{\downarrow}, G) = 0$ and $\mathrm{Pres}(\mathcal{R}{\downarrow}, G) = \mathrm{size}(G) - \Delta(\mathcal{R}{\downarrow}, G)$. □

Fact 1. Let $\mathcal{R}$ is the rule set in Construction 1. For constraints of the form $\exists C$, `Try` $\mathcal{R}$ is a terminating, maximally preserving repair program.

[8] Undirected edges represent two directed edges in opposite direction.

Proof. By Theorem 1, $\texttt{Try}\ \mathcal{R}$ is terminating, correct (If $G \Rightarrow_{\texttt{Try}\ \mathcal{R}} H$, then, by inspection of $\mathcal{R}$, $G \Rightarrow_{\mathcal{R}} H \models \exists\, C$ or $H \cong G \models \exists\, C$), and maximally preserving.□

3.2 Constraints of the form $\exists\,(A, \nexists\, C)$

In this subsection, we consider constraints of the form $\exists\,(A, \nexists\, C)$, with $A \ncong C$, informally meaning that there exists an occurrence of the graph A, but, at that place, not an occurrence of the supergraph C. For these constraints, there is an easy and intuitive program $\{P_a, \langle C \Rightarrow B\rangle\downarrow\}$ where the first subprogram is a repair program for $\exists\, A$ and the second subprogram replaces an occurrence of the graph C by proper subgraphs B of C including A, as long as possible.

To get a more specific repair program, we check whether the constraint $\nexists\,(A, \nexists\, C)$ is satisfied, we "mark" an occurrence of A, apply the "marked" ruleset as long as possible, and, finally, "unmark" the occurrence. We simulate relabelling by adding loops to nodes carrying the marking and replace edges by edges.

Definition 10 (selection). Let $d = \exists\,(A, \nexists\, C)$ be a constraint, $A \subseteq B \subset C$ graphs. Then B_A and C_A denote the A-*marked* versions, i.e., the graph in which all items in the subgraph A are additionally labelled by the name of the item. For a rule $p = \langle C \Rightarrow B\rangle$, $p_A = \langle C_A \Rightarrow B_A\rangle$ is the *marked* rule, and, for a rule set $\mathcal{R}$, $\mathcal{R}_A = \{p_A \mid p \in \mathcal{R}\}$ is the *marked* rule set. The A-*restricted* program

$$\mathcal{R}{\downarrow_A} = \langle \texttt{Check}; \texttt{Select}_A; \mathcal{R}_A\downarrow; \texttt{Unselect}_A\rangle$$

is the program where $\texttt{Check} = \langle \emptyset \Rightarrow \emptyset, \neg d\rangle$, $\texttt{Select}_A = \langle A \Rightarrow A_A\rangle$, $\mathcal{R}_A$ is the marked version of $\mathcal{R}$, and $\texttt{Unselect}_A$ is the inverse rule[9] of $\texttt{Select}_A$.

Remark. The number of steps to obtain a satisfying graph depends from the chosen matches. E.g., for a constraint of the form $\exists\,(A, \nexists\, C)$, there might be some matches, where many occurrences of C have to be removed and matches with only one occurrence of C. For some matches, "dangling edges" have to be removed and for others, there are no dangling edges.

Theorem 2. For constraints of the form $\exists\,(A, \nexists\, C)$ with $A \ncong C$, a terminating, maximally preserving repair program can be effectively constructed.

Construction 2. For a constraint $\exists\,(A, \nexists\, C)$ with $A \ncong C$, let $\{P_a, \mathcal{R}{\downarrow_A}\}$ be the choice program where P_a is the repair program for $\exists\, A$ according to Fact 1 and $\mathcal{R}{\downarrow_A}$ is the A-restricted program with

$$\mathcal{R} := \{\langle C \Rightarrow B\rangle \mid C \supset B \supseteq A \text{ and } (*)\}.$$

where (*) $|\mathrm{E}_C| = |\mathrm{E}_B| + 1$ or $|\mathrm{E}_C| = |\mathrm{E}_B|$ and $|\mathrm{V}_C| = |\mathrm{V}_B| + 1$.

[9] For a plain rule $p = \langle L \Rightarrow R\rangle$, $p^{-1} = \langle R \Rightarrow L\rangle$ is the *inverse* rule of p.

Remark. The rule set $\mathcal{R}$ is — up to isomorphism — a finite, non-empty set of deleting rules. By $B \subset C$, the rule set $\mathcal{R}$ does not contain identical rules. The requirement (*) guarantees that as few nodes as necessary are deleted. Consider, for example, the constraint $\exists\,(\underset{1}{\bullet}\quad\underset{2}{\bullet}, \nexists\,(\underset{1}{\bullet}\!\rightarrow\!\underset{2}{\bullet}\ \bullet))$, meaning that there are two nodes, but no connecting edge and an additional node. In this case, a repair program can delete an edge instead of deleting a node.

Example 4. For 1place $= \nexists\,(\boxed{\text{Pl}}\ \boxed{\text{Pl}})$, the program $\langle \boxed{\text{Pl}}\ \boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}} \rangle\downarrow$ is a terminating repair program. The program works as follows: Whenever there are two places, one place together with its incident edges is deleted, as long as possible.

Proof (of Theorem 2). Let P be the program according to Construction 2.

P is terminating: By construction, P_a is terminating. Moreover, each rule in $\mathcal{R}$ is deleting. Thus, for every finite graph G, there is no infinite transformation.

P is correct: Let $G \Rightarrow_P H$. Application of the program P_a to G yields graph satisfying $\exists\, A$ and $\exists\,(A, \nexists\, C)$. Application of the program $\mathcal{R}\downarrow_A$ to G yields a graph H satisfying $\exists\,(A, \nexists\, C)$: By the semantics of $\downarrow$, no rule of $\mathcal{R}_A$ is applicable to H_A (see below), i.e., for every rule $\langle C_A \Rightarrow B_A\rangle$ in $\mathcal{R}_A$, there is no injective morphism $q_A\colon C_A \hookrightarrow H_A$ and no injective morphism $q\colon C \hookrightarrow H$, such that $q \circ a = h$, i.e., $h \models \nexists\,(A \hookrightarrow C)$ and $H \models \exists\,(A, \nexists\, C)$.

$$\begin{tikzcd}
 & & A \arrow[dl, hook', "a"'] \arrow[dr, hook, "b"] \arrow[dddll, bend right=40, hook', "h"'] & \\
 & C \arrow[rr, Rightarrow] \arrow[d, Rightarrow] \arrow[ddl, bend right, hook', "q"', "/" marking] \arrow[d, phantom, "=", shift right=12] & & B \arrow[d, Rightarrow] \\
 & C_A \arrow[rr, Rightarrow] \arrow[d, hook, "q_A"'] & & B_A \\
H \arrow[r, Rightarrow, "\texttt{Select}_A"'] & H_A & &
\end{tikzcd}$$

P is maximally preserving: The programs $P_a, \texttt{Check}, \texttt{Select}_A, \texttt{Unselect}_A$ preserve the items of the graph G. The number of repairs is bounded by the number of extentions of $p\colon A \hookrightarrow G$, i.e., (*) $\text{pres}(t) \geq \text{size}(G) - |\mathit{Ext}(p)|$. By definition of $\Delta(G,d)$, for each morphism $p\colon A \hookrightarrow G$, $\Delta(G,d) \leq |\text{Ext}(p)|$ and (**) $\text{size}(G) - \Delta(G,d) \geq \text{size}(G) - |\text{Ext}(p)|$. Thus, we have

$$\begin{aligned}
\text{Pres}(P,G) &= \max\nolimits_{t' \in T} \text{pres}(t') && (\text{Def of Pres}(P,G))\\
&\geq \text{pres}(t) && (\text{Def of max})\\
&\geq \text{size}(G) - |\text{Ext}(p)| && (*)\\
&\geq \text{size}(G) - \Delta(G,d) && (**)
\end{aligned}$$

□

Example 5. Let $d = \exists\,(\underset{1}{\bullet}, \nexists\ \underset{1}{\bullet}\rightleftarrows\bullet)$. For the graph $G = \underset{1}{\bullet}\overset{\rightarrow}{\underset{\rightarrow}{\longleftarrow}}\bullet$ and the morphism p form $\underset{1}{\bullet}$ to the node 1 in G, there are two extensions q from $\underset{1}{\bullet}\rightleftarrows\bullet$

to G, thus $\Delta(G,d) = 2$. There are transformations $t : G \Rightarrow_P H$ to a graph H satisfying d with $\text{pres}(t) = \text{size}(G) - 2$ as well as one with $\text{pres}(t) = \text{size}(G) - 1$, i.e., $\text{Pres}(P,G) = \text{size}(G) - 1 > \text{size}(G) - \Delta(G,d)$.

Fact 2. Let $\mathcal{R}$ be the rule set in Construction 2. For constraints of the form $\nexists\, C$, $\mathcal{R}{\downarrow}$ is a terminating, maximally preserving repair program.

Proof. By Theorem 2 and the equivalence $\nexists\, C \equiv \exists\,(\emptyset, \nexists\, C)$, $\mathcal{R}\downarrow_\emptyset = \mathcal{R}\downarrow$ is a terminating, maximally preserving repair program for $\nexists\, C$. □

Fact 3. For all satisfiable, non-valid atomic constraints of nesting depth ≤ 2, there are terminating, maximally preserving repair programs (see Table 1).

constraint	program	
$\exists\, C \equiv \forall(\emptyset, \exists\, C)$	`Try` $\mathcal{R}$	Fact 1
$\nexists\, C \equiv \exists\,(\emptyset, \nexists\, C)$	$\mathcal{R}^+{\downarrow}$	Fact 2
$\exists\,(A, \exists\, C) \equiv \exists\, C$	`Try` $\mathcal{R}$	Fact 1
$\exists\,(A, \nexists\, C)$	$\{P_a, \mathcal{R}^+{\downarrow}_A\}$	Thm 2
$\nexists\,(A, \exists\, C) \equiv \nexists\, C$	$\mathcal{R}^+{\downarrow}$	Fact 2
$\nexists\,(A, \nexists\, C) \equiv \forall(A, \exists\, C)$	$\mathcal{R}{\downarrow}$	Thm 1

Table 1. Satisfiable, non-valid constraints of nesting depth ≤ 2 with repair programs

3.3 Conjunctive constraints

For conjunctive constraints, we make use of the divide and conquer method:

(1) Given a constraint, transform it into an equivalent one in normal form.
(2) Construct repair programs for the subconditions.
(3) Compose the repair programs for the subconditions to a repair program for the more complex condition.

Repair programs P_i for constraints d_i $(i = 1,2)$ can be sequentially composed to a repair program $\langle P_1; P_2 \rangle$ provided that P_2 preserves the constraint d_1.

Definition 11 (constraint preservation). A program P *is d-preserving* if for every transformation $G \Rightarrow_P H$, $G \models d$ implies $H \models d$.

Lemma 2 (Repair). If P_i are terminating repair programs for d_i $(i = 1,2)$ and P_2 is d_1-preserving, then $\langle P_1; P_2 \rangle$ is a terminating repair program for $d_1 \wedge d_2$.

Proof. By termination of the programs P_1 and P_2, the program $\langle P_1; P_2\rangle$ is terminating. Since P_i is a repair program for d_i $(i = 1, 2)$ and P_2 is d_1-preserving, for every transformation $G \Rightarrow_{P_1} H \Rightarrow_{P_2} M$, $H \models d_1$ and $M \models d_1 \wedge d_2$, i.e., $\langle P_1; P_2\rangle$ is a repair program for $d_1 \wedge d_2$. $\square$

There are examples where P_i are repair programs for d_i $(i = 1, 2)$, but neither P_2 is d_1-preserving nor P_1 is d_2-preserving.

Example 6. Consider the constraints alltok $= \forall(\boxed{\text{Pl}}, \exists\,(\boxed{\text{Pl}} \xrightarrow{\text{token}} \boxed{\text{tk}}))$ and tok2places $= \nexists\,(\boxed{\text{Pl}} \xrightarrow{\text{token}} \boxed{\text{tk}} \xleftarrow{\text{token}} \boxed{\text{Pl}})$, meaning that each place has a token but a token does not have two places. There are repair programs $\mathcal{R}_1\!\downarrow$ for alltok (see Example 2) and $\mathcal{R}_2\downarrow$ for tok2places where $\mathcal{R}_2$ contains the rule $\langle \boxed{\text{Pl}} \xrightarrow{\text{token}} \boxed{\text{tk}} \xleftarrow{\text{token}} \boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}}\ \boxed{\text{tk}} \xleftarrow{\text{token}} \boxed{\text{Pl}}\rangle$. Then $\mathcal{R}_2\downarrow$ does not preserve alltok and vice versa. By Lemma 2, we cannot conclude that $\langle \mathcal{R}_1\downarrow; \mathcal{R}_2\downarrow\rangle$ is a repair program for the conjunctive constraint $d =$ alltok $\wedge$ tok2places.

If every rule in P is constraint-preserving, then P is constraint-preserving. Moreover, constraint preservation can be characterized with the help of strongest postconditions. For the definition and construction see e.g. [HP09].

Lemma 3 (characterization). A program P is d-preserving iff the strongest postcondition $\text{sp}(P, d)$ implies d.

Proof. "$\Rightarrow$": If P is d-preserving, then, for every transformation $G \Rightarrow_P H$, $G \models d$ implies $H \models d$. By definition of the strongest postcondition, $H \models \text{sp}(P, d)$. Consequently, $\text{sp}(P, d)$ implies d. '$\Leftarrow$": If $\text{sp}(P, d)$ implies d, then $G \Rightarrow_P H$, $G \models d$ implies $H \models \text{sp}(P, d) \Rightarrow d$, i.e. $H \models d$. $\square$

Remark. For repair programs, one may believe the following: Given repair programs P_i for d_i $(i = 1, 2)$, the program $\langle P_1; P_2^{d_1}\rangle$ where $P_2^{d_1}$ is the program of d_1-preserving rules of P_2, is a repair program. Unfortunately, in general, this is not true because $P_2^{d_1}$ is not a repair program for the constraint d_2.

Example 7. Consider the constraints from example 6, with the respective repair programs $\mathcal{R}_1\downarrow$ for alltok and $\mathcal{R}_2\downarrow$ for tok2places. The tok2places-preserving repair program for alltok is $\mathcal{R}_1'\downarrow$, where $\mathcal{R}_1$ contains the rule $\langle \boxed{\text{tk}}\ \boxed{\text{Pl}} \Rightarrow \boxed{\text{tk}} \xrightarrow{\text{token}} \boxed{\text{Pl}}, \nexists\,(\boxed{\text{Pl}}\ \boxed{\text{tk}} \xleftarrow{\text{token}} \boxed{\text{Pl}}\ ) \wedge \ldots\rangle$. The repair program adds an edge between a place and a token, only if the token has no place. By the application condition, $\mathcal{R}_1'$ is not applicable to arbitrary graphs, thus $\langle \mathcal{R}_2\downarrow; \mathcal{R}_1'\!\downarrow\rangle$ is not a repair program for tok2places $\wedge$ alltok.

Remark. If $\mathcal{R}_i\!\downarrow$ is a repair program for d_i $(i = 1, 2)$ the program $\langle \mathcal{R}_1\!\downarrow; \mathcal{R}_2\!\downarrow\rangle$ is a repair program for $d_1 \wedge d_2$ provided P_2 is d_1-preserving. The program $\mathcal{R}\!\downarrow$ built from the union $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ of the rule sets is a repair program for $d_1 \wedge d_2$ provided that more restrictive requirements are satisfied.

3.4 Disjunctive constraints

Every repair program for a conjunctive constraint is also a repair program for the corresponding disjunctive constraint.

Lemma 4 (Repair).

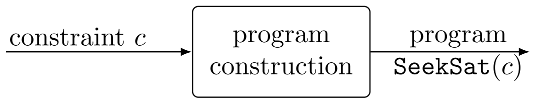
1. If P is a repair program for d and $d \Rightarrow d'$, then P is a repair program for d'.
2. If P_i are repair programs for d_i $(i = 1, 2)$, then P_1, P_2, and $\{P_1, P_2\}$ are repair programs for $d_1 \vee d_2$.

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 statement 1, P_i $(i = 1, 2)$ is a repair program for $d_1 \vee d_2$. By termination of P_1 and P_2, the program $\{P_1, P_2\}$ is terminating. Since P_i is a repair program for d_i $(i = 1, 2)$, for every transformation $G \Rightarrow_{P_i} H$, $H \models d_i$, i.e., $\{P_1, P_2\}$ is a repair program for $d_1 \vee d_2$. □

4 Related concepts

In this section, we present some related concepts of rule-based programs generated from a constraint. We compare proven correctness, completeness, in the sense that for all first-order constraints a program can be automatically generated, and termination of the program for all cases.

In Pennemann [Pen09], an algorithm is given that generates for each graph condition c a non-determistic program `SeekSat`(c), which will find a valid graph for every satisfiable condition. Starting from the empty graph, the algorithm adds items, progressing to a valid graph which satisfies the constraint. Since negative conditions are refuted, the program needs backtracking. The algorithm is correct and complete, but it is not guaranteed to terminate in general. For the non-nested fragment of conditions, `SeekSat` is guaranteed to terminate.



In Nassar et al. [NKR17], a rule-based approach to support a modeler in automatically trimming and completing EMF models and thereby resolving their cardinality violations is proposed. For that, repair rules are automatically generated from multiplicity constraints imposed by a given meta-model.

The control flow of the algorithm consists of two main phases:

(1) Model trimming eliminates supernumerous model elements.
(2) Model completion adds required model elements.

It is shown that both of the algorithms are correct, and, for fully finitely instantiable type graphs, the model completion algorithm terminates. The rules are designed to respect EMF constraints, which are in general not expressible with nested conditions.

In Nentwich et al. [NEF03] a repair framework for inconsistent distributed UML documents is presented. Given a first order formula, the algorithm automatically creates a set of repair actions from which the user can choose, when an inconsistency occurs. These repair actions can either delete, add or change a model document. It can be shown, that the repair actions are correct and complete. The problem of repair cycles is left for future work. Since, in general, it is undecidable, if a constraint is satisfiable, the algorithm may not terminate.

In Mens et al. [PSM15], a regression planner is used to automatically generate sequences of repair actions that transform a model with inconsistencies to a valid model. The initial state of the planner is the invalid model, represented as logical formula, the accepting state is a condition specifying the abscence of inconsistencies. Then, a recursive best-first search is used to find the best suitable plan for resolving the inconsistencies. The correctness of the algorithm is not proven, but the approach is evaluated through tests on different UML models.

	correctness	completeness	termination
this paper	+	-	+
[Pen09]	+	+	-
[NKR17]	+	-	-
[NEF03]	+	+	-
[PSM15]	(+)	?	?

5 Conclusion

In this paper, we have focused on constraints of nesting depth ≤ 2 and have presented terminating repair programs

(1) for all satisfiable constraints without conjunctions & disjunctions (Thms 1, 2)
(2) for conjunctions $d_1 \wedge d_2$ provided that there are terminating repair programs P_i for d_i $(i = 1, 2)$ and one program preserves the other constraint (Lemma 2)
(3) for disjunctions $d_1 \vee d_2$ provided that there is a terminating repair program P_i for d_i for some $i \in \{1, 2\}$ (Lemma 4).

Essential is the following:

(1) The repair programs are directly derived from the constraints.
(2) Dangling edges are deleted before a rule is applied.
(3) **Correctness**. Correctness is proven.
(4) **Termination**. If we look for termination, we get cycles.
If we try to avoid cycles, we get non-termination.

(5) **Maximal Preservation** for atomic constraints of nesting depth ≤ 2.

Further topics could be

(1) Generalization to constraints of arbitrary nesting depth.
(2) Generalization to repair programs with restricted set of input graphs, e.g. described by a constraint.
(3) Construction of a set of repair programs instead of one (as e.g. in [NEF03]).
(4) Use of graph repair for model repair. In particular, what can be done to get a (terminating) repair program that does not create cycles?

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

References

BET12. Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent emf model transformations by algebraic graph transformation. *Software and System Modeling*, 11(2):227–250, 2012.

EEGH15. Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. Springer, 2015.

EEPT06. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer, 2006.

HP09. Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19:245–296, 2009.

MGC13. Nuno Macedo, Tiago Guimarães, and Alcino Cunha. Model repair and transformation with echo. In *Automated Software Engineering, (ASE 2013)*, pages 694–697. IEEE, 2013.

NEF03. Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Software Engineering*, pages 455–464. IEEE Computer Society, 2003.

NKR17. Nebras Nassar, Jens Kosiol, and Hendrik Radke. Rule-based repair of emf models: Formalization and correctness proof. In *Graph Computation Models (GCM 2017)*, 2017. `https://www.uni-marburg.de/fb12/arbeitsgruppen/swt/forschung/publikationen/2017/NKR17.pdf`.

Pen09. Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.

PSM15. Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Resolving model inconsistencies using automated regression planning. *Software and System Modeling*, 14(1):461–481, 2015.

RAB$^+$18. Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Grabriele Taentzer. Translating essential OCL invariants to nested graph constraints for generating nstances of meta-models. *Science of Computer Programming*, 152:38–62, 2018.