International Journal on Software Tools for Technology Transfer (2021) 23: 369–410
https://doi.org/10.1007/s10009-020-00584-x

GENERAL

Special Issue: FASE 2019

A logic-based incremental approach to graph repair featuring delta preservation

Sven Schneider[1] · Leen Lambers[1] · Fernando Orejas[2]

Published online: 7 July 2021

Abstract
We introduce a logic-based incremental approach to graph repair, generating a sound and complete (upon termination) overview of least-changing graph repairs from which a user may select a graph repair based on non-formalized further requirements. This incremental approach features delta preservation as it allows to restrict the generation of graph repairs to delta-preserving graph repairs, which do not revert the additions and deletions of the most recent consistency-violating graph update. We specify consistency of graphs using the logic of nested graph conditions, which is equivalent to first-order logic on graphs. Technically, the incremental approach encodes if and how the graph under repair satisfies a graph condition using the novel data structure of satisfaction trees, which are adapted incrementally according to the graph updates applied. In addition to the incremental approach, we also present two state-based graph repair algorithms, which restore consistency of a graph independent of the most recent graph update and which generate additional graph repairs using a global perspective on the graph under repair. We evaluate the developed algorithms using our prototypical implementation in the tool AutoGraph and illustrate our incremental approach using a case study from the graph database domain.

Keywords Nested graph conditions · Graph repair · Model repair · Consistency restoration · Delta preservation · Graph databases · Model-driven engineering

1 Introduction

Numerous approaches on model inconsistency and repair (see [28] for an excellent recent survey) operate in varying frameworks with diverse assumptions on the underlying model and consistency conditions. In our framework, we consider a typed directed graph (cf. [15]) to be inconsistent if it does not satisfy a given finite set of constraints, which are expressed by graph conditions [19], a specification formalism with the expressive power of first-order logic on graphs. A graph repair in this setting then describes an update of the given graph that results in a graph that is consistent. We consider the more involved problem of deriving a suitable set of graph repairs from which the user then selects the desired graph repair to be applied. Since the set of viable graph repairs is usually infinite, we develop suitable classifications resulting in a finite number of graph repairs. Such a restriction of all graph repairs is already given by only deriving least-changing graph repairs, which do not include other smaller viable graph repairs. The developed graph repair algorithms rely on the model generation technique for graph conditions implemented in the tool AutoGraph from [40].

We consider two scenarios. In the first scenario, which is supported by two state-based graph repair algorithms, the aim is to repair a given graph using one global graph repair, which fixes all problems at once. In the second scenario, which is supported by a delta-based graph repair algorithm, a graph update that changes a given graph to another possibly inconsistent graph is given. Such an inconsistent graph is then to be repaired incrementally using multiple local graph repairs, which fix one problem each. To this end, we track precisely if and how a graph satisfies the consistency constraint by

✉ Sven Schneider
sven.schneider@hpi.de

Leen Lambers
leen.lambers@hpi.de

Fernando Orejas
orejas@lsi.upc.edu

1 Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

2 Universitat Politècnica de Catalunya, Barcelona, Spain

relying on so-called satisfaction trees, which are stored for this purpose in the local state of the process that (a) monitors the current graph, (b) computes graph repairs when the graph is inconsistent and (c) applies a selected graph repair to reestablish consistency. Finally, the delta-based graph repair algorithm allows via a Boolean parameter the generation of only delta-preserving graph repairs, which not only result in a consistent graph no longer violating consistency but which also preserve the changes of the provided graph update by not reverting any of its additions or deletions.

Our contributions are as follows.

- Formal definitions of the key notions of graph updates, graph repairs and graph repair classifications such as *least-changing graph repairs*.
- Formal definitions of two state-based graph repair algorithms and an (incremental) delta-based graph repair algorithm. For these three algorithms, we demonstrate *soundness* (each computed graph repair results in a consistent graph) and *completeness* (upon termination, our algorithms return all desired graph repairs).
- The notion of *satisfaction trees* (STs), which formalizes if and how a graph satisfies a consistency constraint. In particular, we develop (a) the adaptation of an ST when a graph update is applied to the underlying graph and (b) the derivation of violations from an ST, which represent in a detailed way why a consistency constraint is not satisfied.
- Support via parameterization for *delta preservation* in the delta-based case, ensuring—if this feature is selected—that all generated graph repairs preserve the modifications of the most recent graph update that resulted in a consistency-violating graph.

In comparison, most other repair techniques do not provide guarantees for the functional semantics of the graph repairs computed (see conclusion of the survey [28]). With our logic-based graph repair approach, we aim at alleviating this weakness by formally presenting its functional semantics and by describing the details of the underlying graph repair algorithms.
Moreover, while other repair techniques return only one graph repair, we construct a complete set of graph repairs from which the user may select one graph repair based on further requirements.

This paper represents a considerable extension of our previous work presented in [41]. In particular, we added the following contributions. The paper is now *self-contained* by (a) reintroducing also preliminaries on typed directed graphs together with categorical notions used throughout the paper, (b) providing known results on the reasoning approach that is used for the generation of models for graph conditions in our graph repair algorithms, and (c) presenting proofs for all theorems in an appendix. We have added further *examples as well as explanations* to the technical contributions throughout the paper. Moreover, in addition to a running example used for demonstration purposes, we illustrate our approach on a *case study from the graph database domain* and evaluate our algorithms based on a novel prototypical implementation. Technically, we added details (a) for the notion of *isomorphic graph updates* as we compute graph repairs up to isomorphism, (b) for the notion of a *reduction in a graph update* concisely describing graph updates that are smaller than a given graph update with the same effect, and (c) for both *state-based graph repair algorithms*, which were described informally in [41], also including a mechanism to obtain only least-changing graph repairs without a posteriori filtering. We formalized the notion of *violations of a satisfaction tree* to precisely determine and formally cover precise reasons for the non-satisfaction of a given consistency constraint. Finally, we added *support to generate only delta-preserving graph repairs* in the delta-based case, which required substantial modifications of the underlying notions in the incremental approach.

The paper is organized as follows. We introduce preliminaries in Sect. 2 for typed graphs, in Sect. 3 for the employed graph logic on typed graphs, and in Sect. 4 for the reused model generation procedure for the graph logic. Afterwards, we proceed in Sect. 5 by defining graph updates and graph repairs. In Sect. 6, we formally introduce and discuss two state-based graph repair algorithms. We continue with introducing satisfaction trees in Sect. 7, which are needed for the delta-based graph repair algorithm featuring delta preservation in Sect. 8. We evaluate and compare the developed algorithms using our prototypical implementation and discuss matters of resource requirements in Sect. 9. In Sect. 10, we illustrate our approach using a case study from the graph database domain. Finally, we close with a comparison with related work in Sect. 11 and conclusion with outlook in Sect. 12. For proofs of theorems, we refer to "Appendix A".

2 Typed graphs

We provide a self-contained definition of the well-known formalism of typed graphs (see e.g. [15] for an in-depth introduction) including our notation used subsequently. We introduce typed graphs by first introducing plain graphs and plain graph morphisms for the untyped case. Plain graphs contain two sets of nodes and edges and two mappings associating to each edge its source and target node. In this formalization, two edges may have the same source and target, which means that we permit parallel edges. See Fig. 1 for an example of a plain graph (top left).

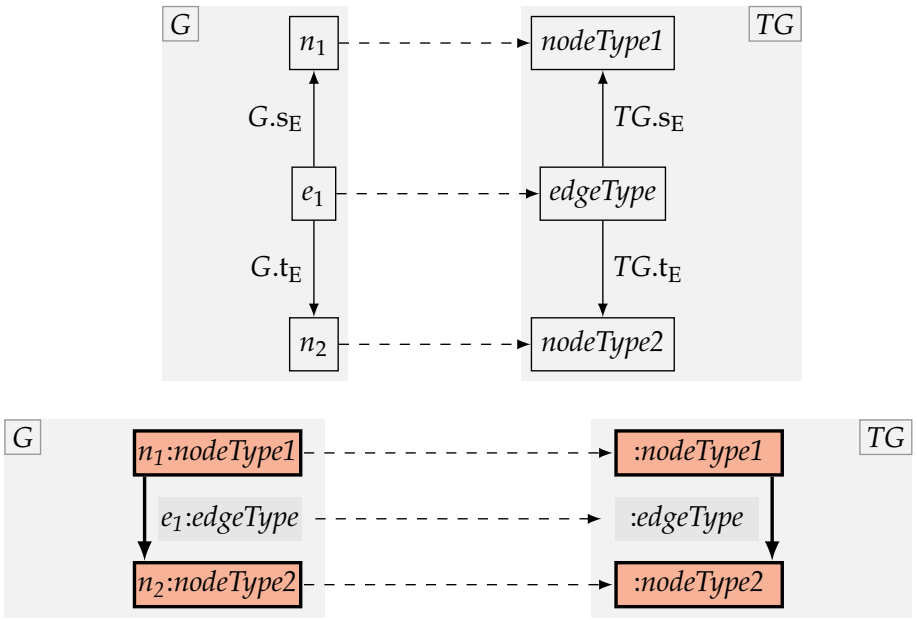


Fig. 1 The plain graph G (top left) and a plain type graph TG (top right), a typing morphism $\tau : G \longrightarrow TG$ (dashed arrows), and the use of our simplified notation (bottom) for G, TG, and τ

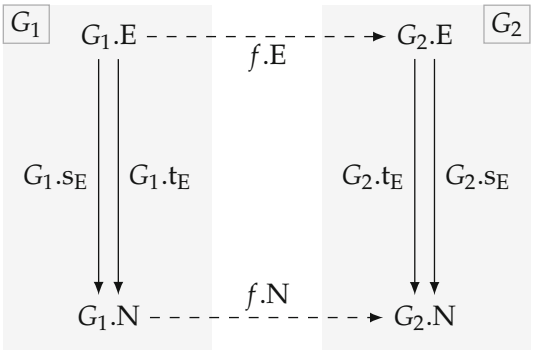


Fig. 2 Plain graphs G_1 and G_2 (solid arrows) and plain graph morphism $f : G_1 \longrightarrow G_2$ (dashed arrows) with their components

Definition 1 (Plain Graph)
If (see Fig. 2 for a visualization) $G.\mathrm{N}$ and $G.\mathrm{E}$ are two disjoint sets of nodes and edges (i.e., they satisfy $G.\mathrm{N} \cap G.\mathrm{E} = \varnothing$), $G.s_E : G.\mathrm{E} \longrightarrow G.\mathrm{N}$ and $G.t_E : G.\mathrm{E} \longrightarrow G.\mathrm{N}$ assign source and target nodes to each edge, and $G = (G.\mathrm{N}, G.\mathrm{E}, G.s_E, G.t_E)$, then G is a *plain graph*, written $G \in \mathcal{S}^{\text{graphs}}$.
Moreover, we define the following abbreviation.

- $\mathcal{S}^{\text{graphs}}_{\text{fin}} = \{G \in \mathcal{S}^{\text{graphs}} \mid \text{finite}(G.\mathrm{N} \cup G.\mathrm{E})\}$ contains all finite plain graphs.

Plain graph morphisms between plain graphs are then given by two maps between the corresponding sets of nodes and edges. A visualization of the required compatibility with the source and target functions for edges is also given in Fig. 1 (top).

Definition 2 (Plain Graph Morphism)
If
- G_1 and G_2 are typed graphs from $\mathcal{S}^{\text{graphs}}$,
- $f.\mathrm{N} : G_1.\mathrm{N} \longrightarrow G_2.\mathrm{N}$,
- $f.\mathrm{E} : G_1.\mathrm{E} \longrightarrow G_2.\mathrm{E}$,
- $f.\mathrm{N} \circ G_1.s_E = G_2.s_E \circ f.\mathrm{E}$,
- $f.\mathrm{N} \circ G_1.t_E = G_2.t_E \circ f.\mathrm{E}$ and
- $f = (f.\mathrm{N}, f.\mathrm{E})$,

then f is a *plain graph morphism from* G_1 *to* G_2, written $f : G_1 \longrightarrow G_2$.

The binary composition of two of these plain graph morphisms is defined as usual for both components of nodes and edges.

Definition 3 (Binary Composition for Plain Graph Morphisms)
If $f_1 : G_1 \longrightarrow G_2$, $f_2 : G_2 \longrightarrow G_3$, and $f_3 : G_1 \longrightarrow G_3$ are plain graph morphisms and, moreover, $f_3.\mathrm{N} = f_2.\mathrm{N} \circ f_1.\mathrm{N}$ and $f_3.\mathrm{E} = f_2.\mathrm{E} \circ f_1.\mathrm{E}$, then f_3 is the *composition of* f_2 *and* f_1, written $f_3 = f_2 \circ_{\mathrm{p}} f_1$.

Technically, the typing of typed graphs is formalized by means of an additional plain graph morphism that has a type graph TG as a target. See Fig. 1 for an example of a typed graph, a typing morphism, a type graph, and the simplified notation for typed graphs that we use in the remainder.

Definition 4 (**Typed Graph**)
If $\tau : G \longrightarrow TG$ is a plain graph morphism, then τ is also a *typed graph over TG*, written $\tau \in \mathcal{S}_{TG}^{\text{graphs}}$.
Moreover, we define the following abbreviation.

- $\mathcal{S}_{\text{fin},TG}^{\text{graphs}} = \{\tau : G \longrightarrow TG \mid G \in \mathcal{S}_{\text{fin}}^{\text{graphs}}\}$ contains all finite graphs typed over TG.

Morphisms between typed graphs are then required to preserve the typing for all elements.

Definition 5 (**Typed Graph Morphism**)
If $\tau_1 : G_1 \longrightarrow TG \in \mathcal{S}_{TG}^{\text{graphs}}$, $\tau_2 : G_2 \longrightarrow TG \in \mathcal{S}_{TG}^{\text{graphs}}$, and $f : G_1 \longrightarrow G_2$ are plain graph morphisms and, moreover, $\tau_2 \circ f = \tau_1$, then f is a *(typed) graph morphism from* τ_1 *to* τ_2, written $f : \tau_1 \longrightarrow \tau_2$.

We define the binary composition of typed graph morphisms along the lines of the composition of plain graph morphisms.

Definition 6 (**Binary Composition for Typed Graph Morphism**)
If $f_1 : \tau_1 \longrightarrow \tau_2$, $f_2 : \tau_2 \longrightarrow \tau_3$, and $f_3 : \tau_1 \longrightarrow \tau_3$ are typed graph morphisms and, moreover, $f_3 = f_2 \circ_{\text{p}} f_1$, then f_3 is the *composition of* f_2 *and* f_1, written $f_3 = f_2 \circ f_1$.

To ease presentation, we handle typing of graphs and typed graph morphisms implicitly in the remainder of the paper and refer to typed graphs as graphs and to typed graph morphisms as morphisms.

A morphism $f : A \longrightarrow B$ is an inclusion morphism, if f.N and f.E are inclusions, which is denoted by $f = \text{inc}(A, B)$. A morphism $f : A \longrightarrow B$ is an identity morphism, if f is an inclusion morphism and $A = B$, which is denoted by $f = \text{id}(A)$.

Typed graphs as introduced here with morphisms, the composition operation and identity morphisms determine a category.

Theorem 1 (**Typed Graphs are a Category**)
If Ob is the class of graphs from Definition 4, Mor(A, B) *is the set of morphisms of type* $A \longrightarrow B$ *from Definition 5,* $\circ$ *is the binary composition operation from Definition 6, and* $\text{id}(A)$ *is the unique identity morphism, then* TGraphs $=$ $(Ob, Mor, \circ, \text{id})$ *is a category.*

See Fig. 3 for a class diagram representing a type graph that is used later on in Sect. 10 in the context of our case study. Since the inheritance relations of this class diagram are not directly supported in typed graphs, we flatten the inheritance hierarchy in our examples using this class diagram. Note that type graphs serve as natural formalization of the notion of class diagrams as demonstrated in previous literature mainly from the graph transformation domain, see e.g. [2,25,36]. This relationship is in particular studied more extensively also in the context of the Eclipse Modeling Framework in [9].

We now discuss some categorical notions and constructions for the category TGraphs of typed graphs used throughout the paper.

The empty graph, which has no nodes or edges, is denoted Ø. Also, the empty graph is initial and therefore there is a unique morphism of type $\emptyset \longrightarrow G$ denoted by $\text{i}(G)$ to any graph G.

A morphism $f : A \longrightarrow B$ is a monomorphism of TGraphs, if f.N and f.E are injective, which is denoted by $f : A \hookrightarrow B$ or $\text{mono}(f)$. A morphism $f : A \longrightarrow B$ is an epimorphism of TGraphs, if f.N and f.E are surjective, which is denoted by $f : A \twoheadrightarrow B$ or $\text{epi}(f)$. Finally, a morphism $f : A \longrightarrow B$ is an isomorphism of TGraphs, if f.N and f.E are bijective, which is denoted by $f : A \overset{\text{i}}{\hookrightarrow\!\!\!\!\twoheadrightarrow} B$ or $\text{isom}(f)$.

The pushout $(g_1 : B \longrightarrow D, g_2 : C \longrightarrow D)$ of two morphisms $(f_1 : A \longrightarrow B, f_2 : A \longrightarrow C)$, abbreviated subsequently by $\text{PO}(g_1, g_2, f_1, f_2)$, captures on an intuitive level with graph D the union of the two graphs B and C where the morphisms f_1 and f_2 are used to identify common elements in B and C (i.e., only considering the edge component here, $f_1.\text{E}(x)$ and $f_2.\text{E}(x)$ are to be understood equal when computing the union and $g_1.\text{E}(x) = g_2.\text{E}(y)$ means that x and y are identified when constructing the union). See Fig. 4a for an example of a pushout in TGraphs.

The pullback $(f_1 : A \longrightarrow B, f_2 : A \longrightarrow C)$ of two morphisms $(g_1 : B \longrightarrow D, g_2 : C \longrightarrow D)$, abbreviated subsequently by $\text{PB}(f_1, f_2, g_1, g_2)$, captures on an intuitive level with graph A the intersection of the two graphs B and C where the morphisms are used to identify common elements in B and C (i.e., only considering the edge component here, $g_1.\text{E}(x) = g_2.\text{E}(y)$ means that x and y are to be understood equal when computing the intersection and $f_1.\text{E}(x)$ and $f_2.\text{E}(x)$ are identified when constructing the intersection). See Fig. 4b for an example of a pullback in TGraphs.

3 Graph logic GL

Graph logics are used to specify different kinds of graphs in terms of their graph elements, which are the nodes and edges for typed graphs. In this paper, we use the graph logic GL of nested graph conditions, which is equivalent to first-order logic on graphs [14] as shown in [19,35]. Hence, on the one hand, GL is well applicable as it can express many relevant properties but, on the other hand, problems such as satisfiability are undecidable in general (see Sect. 4 where we also discuss an existing semi-decision procedure for this problem). A basic limitation of the first-order logic GL is that transitive reachability cannot be expressed but extensions of GL in this directions are ongoing work [33].

A more expressive and commonly used logic for the specification of graphs is OCL [31], for which partial transla-

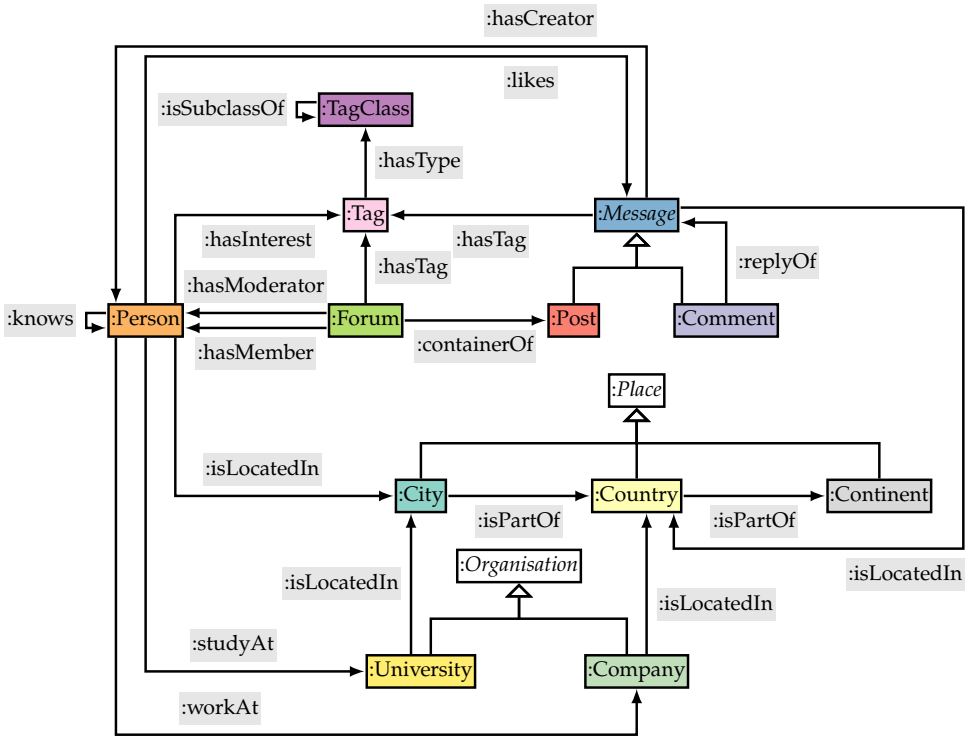


Fig. 3 Class diagram from the social network benchmark [44]

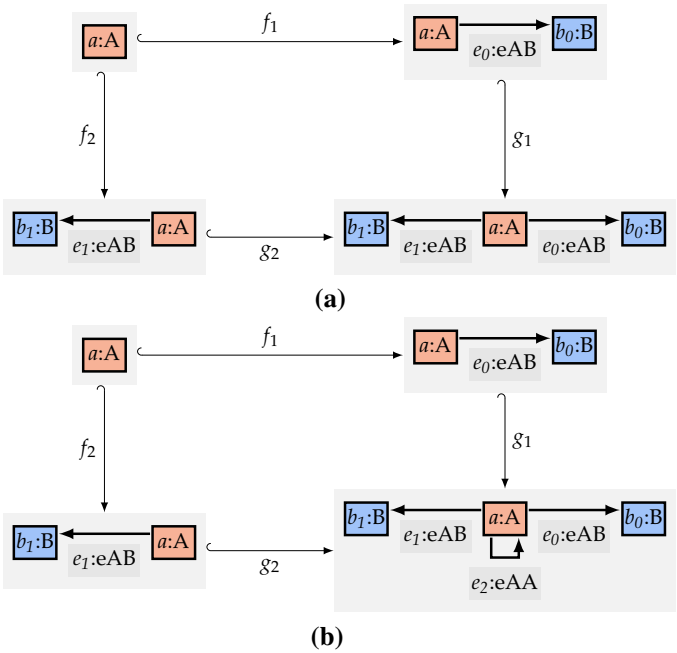


Fig. 4 Examples of pushout and pullback in TGraphs: note that the two diagrams are not identical because f_1 and f_2 are not jointly surjective (i.e., f_1 and f_2 do not map together to all nodes and edges of their common target graph) in **b** whereas f_1 and f_2 are constructed to be jointly surjective in **a**. **a** The pushout construction identifies and therefore overlaps the node a in the target graphs of f_1 and f_2. **b** The pullback construction only includes an :A node because only the a nodes in the source graphs of g_1 and g_2 are mapped to the same node by g_1 and g_2

tions to GL have been considered in [4,29,34]. In particular, Kleppe and Rensink moreover elaborate how typical meta-model or class diagram constraints such as bidirectionality constraints, containment constraints, indexing constraints, containment constraints and multiplicities can be formalized using type graphs in combination with graph constraints. We employ GCs in Sect. 10 in the context of the Social Network Benchmark [44] of the Linked Data Benchmark Council (LDBC) using the type graph presented in Fig. 3 stating desired meta-model constraints. Also see [1] for a survey considering graph repair in the context of graph databases that are expected to satisfy various forms of integrity constraints.

The graph conditions (GCs) of GL are used later on to specify properties on graphs. GCs are constructed inductively using the propositional operators for finite *conjunction*

and *negation* from which further propositional operators can be derived as usual. Moreover, GCs feature an *exists* operator to state facts about the existence or nonexistence of *finite* graph patterns in a possibly infinite given graph, called host graph. These graph patterns are formalized using monomorphisms of which the target graph, which is thereby an extension of the possibly empty source graph, represents the graph pattern. Extensions of patterns are then given again by monomorphisms from one pattern to another. Using only monomorphisms in GCs ensures that the patterns described by graphs cannot shrink in size (non-injective mappings in morphisms merge nodes or edges). For the base case, the empty pattern can be trivially found in a host graph G using the initial morphism $\mathrm{i}(G)$. In general, monomorphisms from graph patterns to the host graph G are called matches.

Definition 7 (Graph Conditions)
If $H \in \mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ is a finite typed graph, then ϕ *is a graph condition over* H, written $\phi \in \mathcal{S}^{\text{GC}}_{TG,H}$, if one of the following items applies.

- $\phi = \wedge S$ and S is a finite set of GCs over H.
- $\phi = \neg\bar{\phi}$ and $\bar{\phi}$ is a GC over H.
- $\phi = \exists(f : H \hookrightarrow H', \bar{\phi})$ and $\bar{\phi}$ is a GC over H'.

Moreover, we define the following abbreviations.

- true: $\top = \wedge\varnothing$
- false: $\bot = \neg\top$
- disjunction: $\vee S = \neg(\wedge\{\neg\bar{\phi} \mid \bar{\phi} \in S\})$
- universal quantification: $\forall(f, \phi) = \neg\exists(f, \neg\phi)$

The abbreviations in the previous definition can also be understood as operators that are derived from the three defined operators $\neg$, $\wedge$, and $\exists$. In the remainder, we only define operations for the defined operators but not for derived operators to avoid cluttering.

In our examples, for improved readability, we only employ inclusion morphisms in GCs and for the case of $\exists(f : H \hookrightarrow H', \phi)$, we visualize the inclusion morphism f by all nodes and edges that are in $H' - H$ or that are connected to such elements. Also, for the delta-based graph repair algorithm in Sect. 8, we require that no isomorphisms are used in the consistency constraint given by a GC. See Fig. 5 for an example of a GC demonstrating the use of nesting and propositional operators.

We now define the set of all subconditions of a GC as follows for later use.

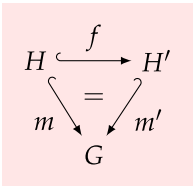
Definition 8 (Subconditions of a Graph Condition)
If $H \in \mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ is a finite typed graph, and $\phi \in \mathcal{S}^{\text{GC}}_{TG,H}$ is a GC over H, then $\mathrm{sub}(\phi) = R$ *is the set of all subconditions of* ϕ, if one of the following items applies.

- $\phi = \wedge S$ and $R = \{\phi\} \cup \{\mathrm{sub}(\bar{\phi}) \mid \bar{\phi} \in S\}$.
- $\phi = \neg\bar{\phi}$ and $R = \{\phi\} \cup \mathrm{sub}(\bar{\phi})$.
- $\phi = \exists(f : H \hookrightarrow H', \bar{\phi})$ and $R = \{\phi\} \cup \mathrm{sub}(\bar{\phi})$.

The satisfaction relation for GL is given below in the form of a recursive definition that follows the inductive definition of GCs. Its definition follows [19] and is as expected for the operators *conjunction* and *negation*. For the case of $\exists(f : \emptyset \hookrightarrow H, \bar{\phi})$, we first consider an extension of the empty pattern given by a monomorphism $f : \emptyset \hookrightarrow H$. For the satisfaction, we then need to be able to find a match $m : H \hookrightarrow G$ into the host graph that also has to satisfy the subcondition $\bar{\phi}$. When $\bar{\phi}$ contains an extension of the pattern H from before using a monomorphism $f' : H \hookrightarrow H'$, we need to be able to find a match $m' : H' \hookrightarrow G$ into the host graph that is an extension of the previous match $m : H \hookrightarrow G$ and that then again satisfies the next-level subcondition $\bar{\phi}'$. This means for m' that it must match all elements according to m w.r.t. the renaming given by f: formally, we must ensure that the new monomorphism m' satisfies $m' \circ f = m$. Note that the satisfaction check for the *exists* operator may not succeed when there is no suitable extension monomorphism m'.

Definition 9 (Satisfaction of Graph Conditions)
If $H \in \mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ is a finite typed graph, $\phi \in \mathcal{S}^{\text{GC}}_{TG,H}$ is a GC over H, and $m : H \hookrightarrow G$ is a match of H in G, then m *satisfies* ϕ, written $m \models_{\text{GC}} \phi$, if one of the following items applies.

- $\phi = \wedge S$ and $\forall\bar{\phi} \in S.\ m \models_{\text{GC}} \bar{\phi}$.
- $\phi = \neg\bar{\phi}$ and not $m \models_{\text{GC}} \bar{\phi}$.
- $\phi = \exists(f : H \hookrightarrow H', \bar{\phi})$ and $\exists m' : H' \hookrightarrow G.\ m = m' \circ f \wedge m' \models_{\text{GC}} \bar{\phi}$.



Also, if $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph and ϕ is satisfied by the initial morphism to G (i.e., $\mathrm{i}(G) \models_{\text{GC}} \phi$), then G *satisfies* ϕ, written $G \models_{\text{GC}} \phi$.

See Fig. 5e for an example of a satisfaction proof for a GC, which follows the nested structure of the GC by applying the satisfaction relation defined above.

Finding matches $m : H \hookrightarrow G$ of a pattern H in a given host graph G according to the satisfaction relation above is NP-complete (note that both graphs G and H vary in typical applications) but the development of static and dynamic heuristics for matching graphs is an active field of research [3,6,8,11,18,23]. For example, if the graphs H and G are connected, then a partial match of H in G can be extended to a match by local extension.

4 Automated reasoning for GL

We now present automated reasoning support for GL in the form of the algorithm $\mathcal{A}$ from [39,40] for which tool support is available in AutoGraph. While satisfiability is undecidable for GL as pointed out before, this problem can be solved for many relevant instances. The algorithm $\mathcal{A}$ takes a GC ϕ as an input and attempts to rewrite ϕ into an equivalent GC ϕ'. The computation of $\mathcal{A}$ may not terminate possibly computing a continuously growing GC. However, if the computation terminates, the resulting condition is of the following restricted form. Firstly, ϕ' is a finite disjunction of GCs of the form $\exists(f : \emptyset \hookrightarrow H, \bar{\phi})$. Secondly, each $\bar{\phi}$ is a finite conjunction of GCs of the form $\neg\exists(f' : H \hookrightarrow H', \bar{\phi}')$ where f' is no isomorphism. For soundness, it is known that the GC ϕ and the resulting GC ϕ' are satisfied by the same graphs, which means that the two conditions are indeed equivalent. Note that during any computation of $\mathcal{A}$, elements of the resulting disjunction are computed incrementally, which means that the condition computed so far at any point in the computation invariantly implies the input condition ϕ.

Moreover, as the main feature of $\mathcal{A}$, it has been shown that each graph H that can be directly obtained from an element $\exists(f : \emptyset \hookrightarrow H, \bar{\phi})$ of the returned disjunction satisfies the given input condition. Also, the finite conjunction $\bar{\phi}$ describes in each case how the graph H can or cannot be extended to graphs $\bar{H}$ still satisfying the given GC. Note that the property that f' is no isomorphism is essential for this extraction of models to ensure that H indeed satisfies the GC. The set of graphs H obtained from the resulting disjunction is by construction complete in the case of termination in the sense that all minimal graphs satisfying the given condition are represented by one element of the disjunction. See Fig. 6a for an example of a GC resulting in a terminating application of $\mathcal{A}$ where the graphs given in Fig. 6b can be obtained from the returned GC and Fig. 6c for a GC resulting in a nonterminating computation. Nevertheless, we point out that the computation performed by $\mathcal{A}$ always proceeds in a reasonable direction (attempting to construct the smallest graphs satisfying the given GC by incrementally enlarging candidates for such a smallest graph) but may not terminate because the smallest graph satisfying a given GC may be an infinite graph, which can't be generated by incrementally adding a finite number of elements.

From the computation of such minimal graphs satisfying the GC, we can deduce that a GC is satisfiable when $\mathcal{A}$returns a non-empty disjunction. Moreover, it has been shown that $\mathcal{A}$ terminates and returns the empty disjunction when the GC is not satisfiable meaning that the procedure is refutationally complete. Note that several other problems such as determining useless subconditions, equivalence and entailment can be checked (up to termination of the procedure) as a consequence of the discussed results.

In subsequent sections, we employ the presented algorithm $\mathcal{A}$ for computing the finite set $\mathcal{M}(\phi)$ of all finite minimal models of ϕ.

Definition 10 (Minimal Models)
If $\phi \in \mathcal{S}^{\mathrm{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, then the finite set of finite typed graphs $\mathcal{M}(\phi) \subseteq_{\mathrm{fin}} \mathcal{S}^{\mathrm{graphs}}_{\mathrm{fin},TG}$ satisfies[1]

- *soundness*: each graph in the returned set satisfies the GC (i.e., $G \in \mathcal{M}(\phi)$ implies $G \models_{\mathrm{GC}} \phi$),
- *completeness*: for each graph satisfying the GC, there is a smaller graph in the returned set (i.e., $G_1 \models_{\mathrm{GC}} \phi$ implies that there is some $f : G_2 \hookrightarrow G_1$ for some $G_2 \in \mathcal{M}(\phi)$), and
- *uniqueness*: two different returned graphs cannot be included in each other (i.e., $G_1 \in \mathcal{M}(\phi)$, $G_2 \in \mathcal{M}(\phi)$, and $G_1 \neq G_2$ implies that there is no monomorphism $f : G_1 \hookrightarrow G_2$).

5 Graph updates and graph repairs

We now define graph updates to formalize arbitrary modifications of graphs that are executed by an external process such as a user or another process. Afterwards, we define graph repairs as the desired graph updates that modify a graph such that the resulting graph satisfies a given GC. Moreover, we further classify graph updates and graph repairs by means of desirable properties that should be satisfied.

Arbitrary graph modifications are well known in the domain of graph transformation (see e.g. [15] for a thorough introduction) where graph transformation rules are used to generate such modifications. We abstract here from the concrete procedure that leads to graphs not satisfying a given GC but rely on the following definition in which a graph update of G_1 resulting in a graph G_2 is represented by a span (i.e., a pair of two morphisms with common domain) of two monomorphisms $(\ell : D \hookrightarrow G_1, r : D \hookrightarrow G_2)$. In this span, the graph D represents the part of G_1 that is preserved by the update, the monomorphism ℓ describes the preserved/removed graph elements, and the monomorphism r describes the preserved/added graph elements. In particular, the elements in $\ell(D)$ are preserved, the elements in $G_1 - \ell(D)$ are removed, the elements in $r(D)$ are preserved and the elements in $G_2 - r(D)$ are added. See Fig. 7a for an example of a graph update that deletes and also adds elements.

Definition 11 (Graph Update)
If $\ell : D \hookrightarrow G_1$ and $r : D \hookrightarrow G_2$ are monomorphisms, then (ℓ, r) *is a graph update*, written $(\ell, r) \in \mathcal{S}^{\mathrm{upd}}$.

[1] Here, $A \subseteq_{\mathrm{fin}} B$ means that A is a finite subset of B.

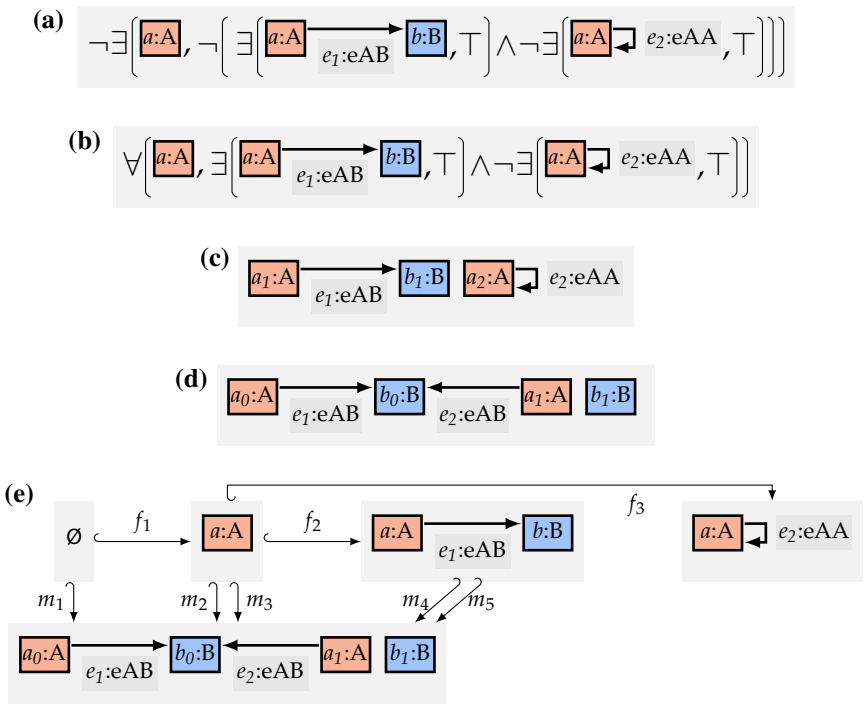


Fig. 5 Example of a GC, a graph, and a satisfaction proof. **a** A GC stating that every node of type :A has an edge of type :eAB to a node of type :B but no self loop of type :eAA. See also **b** for the same condition using the abbreviation for *forall*. **b** The GC from **a** where the abbreviation for *forall* has been used. **c** A graph that does not satisfy the GC from **a** because the node a_2:A has no connected :B node and also a self loop. **d** A graph that satisfies the GC from **a** according to the proof in **e**. **e** For verifying that the graph (called G here) from **d** satisfies the GC (called ϕ here) from **b**, written $G \models_{\text{GC}} \phi$, we prove that $m_1 = \text{i}(G) \models_{\text{GC}} \phi$. We find two possible match morphisms m_2 and m_3 matching the node a to a_0 and a_1. Because of the universal quantification in ϕ, we must consider both. For m_2, we can find an extension m_4 that matches e_1 to e_1 and b to b_0. Also, we do not find an extension of m_2 that matches the self loop on a_0 as required. For m_3, we can find an extension m_5 that matches e_1 to e_2 and b to b_0. Also, we do not find an extension of m_3 that matches the self loop on a_1 as required. This completes the proof and shows that G satisfies ϕ

We now define graph repairs (to be computed in subsequent sections) as those graph updates that result in a graph that satisfies a consistency constraint, which is given in the form of a GC ϕ.

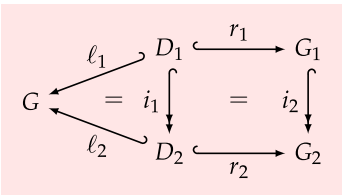
Definition 12 (Graph Repair)
If $(\ell : D \hookrightarrow G_1, r : D \hookrightarrow G_2) \in \mathcal{S}^{\text{upd}}$ is a graph update, $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, and G_2 satisfies ϕ (i.e., $G_2 \models_{\text{GC}} \phi$), then (ℓ, r) *is a graph repair of* G_1 *with respect to* ϕ, written $(\ell, r) \in \mathcal{S}^{\text{repair}}(G_1, \phi)$.

Note that we do not require the input graph G_1 to be inconsistent in this definition, which permits the graph update $(\text{id}(G_1), \text{id}(G_1))$ with the identity morphism on G_1 to be a graph repair as well in this case. See Figs. 7b and 7c for two examples of graph repairs.

We now introduce notions for classifying graph updates and graph repairs. Note that the properties defined for graph updates immediately translate to graph repairs as well.

We define that two graph updates $u_1 = (\ell_1, r_1)$ and $u_2 = (\ell_2, r_2)$ with common input graph are isomorphic when there are two isomorphisms that show that the same modifications are applied in both graph updates up to renaming. The graph repair algorithms that we introduce in Sects. 6 and 8 compute graph repairs up to isomorphism. However, to ease presentation, we avoid a detailed technical handling as usual in the remainder.

Definition 13 (Isomorphic Graph Updates)
If u_1 and u_2 are two graph updates from $\mathcal{S}^{\text{upd}}$, $i_1 : D_1 \overset{\text{i}}{\hookrightarrow} D_2$ and $i_2 : G_1 \overset{\text{i}}{\hookrightarrow} G_2$ are two isomorphisms satisfying $\ell_1 = \ell_2 \circ i_1$ and $i_2 \circ r_1 = r_2 \circ i_1$, then u_1 *and* u_2 *are isomorphic*, written $u_1 \cong u_2$.



Two graph updates $u_1 = (\ell_1, r_1)$ and $u_2 = (\ell_2, r_2)$ describe the same modification when they agree on the input graph and the output graph. In this scenario, we define that u_2 is a reduction of u_1 when (a) the two graph updates obtain their common modification in a compatible way but (b) u_2

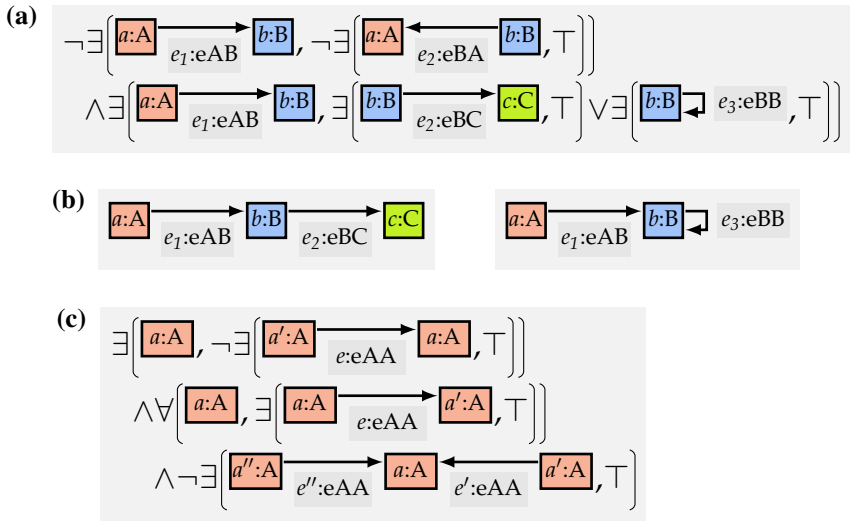
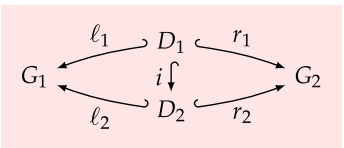


Fig. 6 Example of two GCs and the application of $\mathcal{M}$ and $\mathcal{A}$ for computing minimal graphs for the GCs. **a** A GC ϕ stating that (a) for every edge from an :A node to a :B node there is also an edge in reverse direction and (b) there is an edge from an :A node to a: B node such that (b1) there is also an edge from the :B node to a :C node or (b2) the :B node has a self-loop. **b** The two minimal graphs G_1 and G_2 obtained using $\mathcal{M}(\phi)$ from the GC ϕ from **a**. **c** A GC ϕ formalizing the Peano axioms stating that (a) there is a first :A node without a predecessor, (b) every :A node has a successor, and (c) no :A node has two predecessors. The computation $\mathcal{A}(\phi)$ does not terminate for the GC ϕ as it first constructs a graph with one node of type :A and then incrementally extends this graph adding one additional successor in every step

performs less deletions and additions of nodes and edges. That is, u_1 may delete additional elements (ℓ_1 deletes at least those elements deleted by ℓ_2) but restores the additionally deleted elements afterwards (r_1 adds all elements added by r_2 and also those additionally removed by ℓ_1). Note that the reduced graph update u_2 has a bigger graph D_2 because it preserves more elements.

Definition 14 (Reduction in Graph Update)
If u_1 and u_2 are two graph updates from $\mathcal{S}^{\text{upd}}$, $i : D_2 \hookrightarrow D_1$ is a monomorphism satisfying $\ell_1 = \ell_2 \circ i$, and $r_1 = r_2 \circ i$, then u_2 *is a reduction of* u_1 *according to* i, written $u_2 \subseteq^i u_1$ or simply $u_2 \subseteq u_1$.



Moreover, we define the following abbreviation.

- u_2 *is a strict reduction of* u_1 *according to* i, written $u_2 \subset^i u_1$ or simply $u_2 \subset u_1$, when $u_2 \subseteq^i u_1$ and not $u_1 \subseteq u_2$.

Note that the graph repair presented in Fig. 7b is a strict reduction in the graph repair from Fig. 7c.

We now introduce canonical graph updates, which have a maximal graph D preserving as many nodes and edges as possible from G_1 to G_2, which means that the monomorphism r does not undo deletions of the monomorphism ℓ. For example, for a nonempty graph G, the graph update $u_1 = (\ell : \emptyset \hookrightarrow \text{i}(G), r : \emptyset \hookrightarrow \text{i}(G))$ is noncanonical because it first deletes all elements from G and then restores these elements afterwards using r. In this case, $(\text{id}(G), \text{id}(G))$ is the unique canonical reduction of u_1.

Definition 15 (Canonical Graph Update)
If $u_1 \in \mathcal{S}^{\text{upd}}$ is a graph update and there is no graph update $u_2 \in \mathcal{S}^{\text{upd}}$ that is a strict reduction of u_1 (i.e., $u_2 \subset u_1$), then u_1 *is a canonical graph update*, written $u_1 \in \mathcal{S}^{\text{upd}}_{\text{can}}$.

Note that the graph repair presented in Fig. 7b is canonical.

We state that every graph update can be reduced to a canonical graph update.

Theorem 2 *(Existence of Canonical Graph Update)*
If $u_1 \in \mathcal{S}^{\text{upd}}$ *is a graph update, then there is a canonical graph update* $u_2 \in \mathcal{S}^{\text{upd}}_{\text{can}}$ *that is a reduction of* u_1 *(i.e.,* $u_2 \subseteq u_1$*).*

We now relate two graph updates u_1 and u with the same input graph but different output graphs. In this case, u_1 is a sub-update (see [32] and the similar notion of a derived span in [16, Definition 4.1, p. 44]) of u whenever the modifications defined by u_1 are fully contained in the modifications defined by u. This is the case when every element deleted by u_1 is also deleted by u and every element added by u_1 is also added by u while u is permitted to delete further elements and to add further elements. Technically, u_1 is a sub-update of u, if there is another graph update u_2 such that (a) u_2 has the same output graph as u, (b) u_1 and u_2 can be applied sequentially resulting in the graph update u, and (c) u_2 does not delete any element that was added before by u_1.

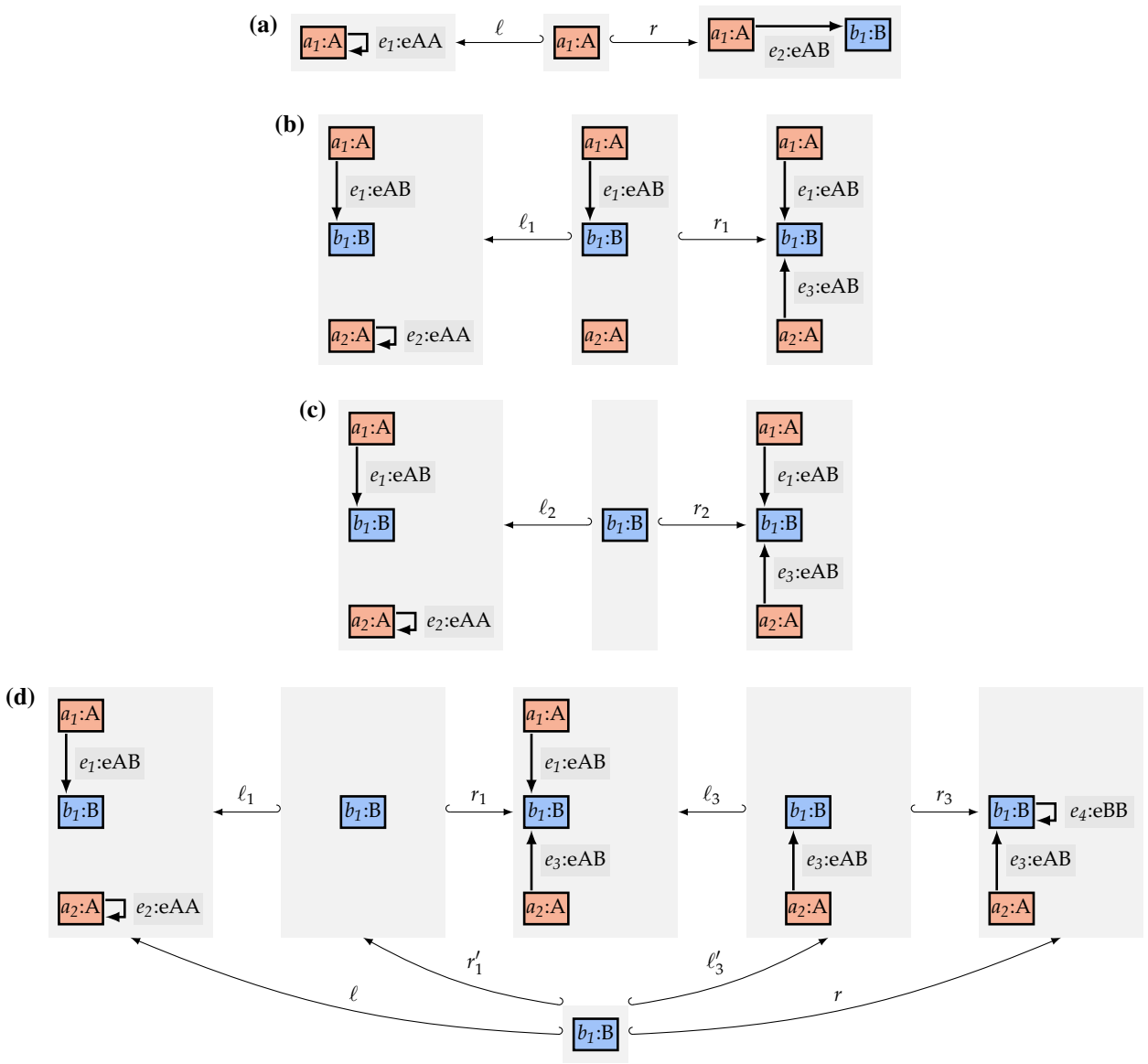


Fig. 7 Examples of graph updates and graph repairs. **a** A graph update that deletes the edge e_1 and then adds node b_1 and edge e_2. **b** A graph repair for the graph from Fig. 5c w.r.t. the GC from Fig. 5a. Also, according to Definition 14, the graph repair is a strict reduction of the graph repair from **c**. Moreover, according to Definition 15, the graph repair is a canonical graph update. Lastly, according to Definition 17, the graph repair is a least changing graph repair. **c** A graph repair for the graph from Fig 5c w.r.t. the GC from Fig 5a. **d** An example of a graph repair (ℓ, r) using the graph repair (ℓ_1, r_1) from **b** as a first step. That is, the graph repair from **b** is a sub-update

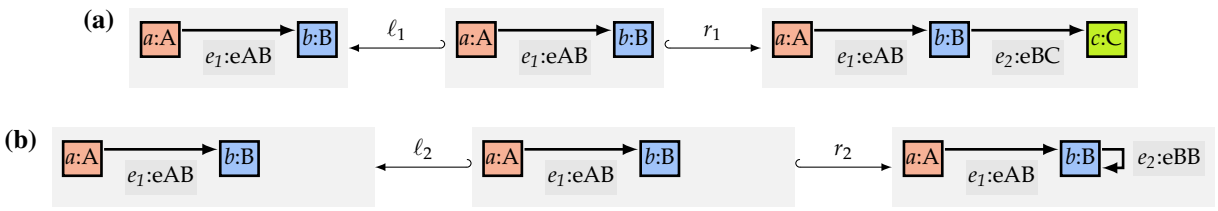
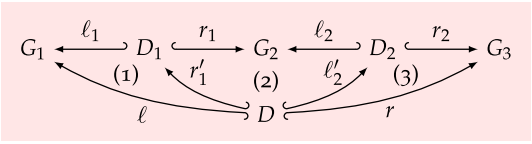


Fig. 8 Comparison of least changing graph repairs and minimal atomic graph repairs. **a** A first least changing graph repair w.r.t. the GC from Fig 6a that is not a minimal atomic graph repair because the graph repair from **b** requires only one atomic edit operation whereas the one depicted here requires two atomic edit operations. **b** A second least changing graph repair w.r.t. the GC from Fig 6a, which requires one atomic edit operation

Definition 16 (**Sub-update** [32])
If $u = (\ell, r)$, $u_1 = (\ell_1, r_1)$, and $u_2 = (\ell_2, r_2)$ are graph updates in $\mathcal{S}^{\text{upd}}$, (1) and (3) commute, and (2) is a pushout and pullback, then u_1 *is a sub-update of* u *w.r.t.* u_2, written $u_1 \leq^{u_2} u$ or $u_1 \leq u$.



Moreover, we define the following abbreviations.

- u_1 *is a strict sub-update of* u, written $u_1 <^{u_2} u$ or $u_1 < u$, when $u_1 \leq^{u_2} u$ and not $u \leq u_1$.
- *The composition of* u_1 *and* u_2, written $u_1 \circ u_2$, is some u satisfying $u_1 \leq^{u_2} u$ (if it exists) and $\bot$ otherwise.

In this definition, the existence of r_1' and the commutation of (1) means that u_1 does not delete more than u, the existence of ℓ_2' and the commutation of (3) means that u_2 does not add more than u, the commutation of (2) means that graph elements in D are equally identified in D_1 and D_2, the requirement that (2) is a pullback means that u_1 and u_2 do not preserve more than u, and the requirement that (2) is a pushout means that u_2 preserves all elements added by u_1. Also note that a graph update u resulting from the composition of u_1 and u_2 does not need to be canonical as r_2 ma add elements that have been deleted by ℓ_1 before.

See Fig. 7d for an example where the graph repair from Fig. 7b is a sub-update of another graph repair.

We now introduce least changing graph repairs, which are those graph repairs for which no strict sub-updates exists (in a given set U of graph repairs) and which already repair the graph at hand. Stated differently, these graph repairs determine successful modifications establishing consistency preserving as many nodes/edges from the input graph as possible compared to the graph repairs in U.

Definition 17 (**Least Changing Graph Repair**)
If $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, $u \in \mathcal{S}^{\text{repair}}(G, \phi)$ is a graph repair of G with respect to ϕ, $U \subseteq \mathcal{S}^{\text{repair}}(G, \phi)$ is a set of graph repairs, and there is no graph update $u' \in U$ that is a strict sub-update of u (i.e., $u' < u$), then u *is a least changing graph repair of* G *w.r.t.* ϕ *and* U, written $u \in \mathcal{S}^{\text{repair}}_{\text{lc}}(G, \phi, U)$.

When $U = \mathcal{S}^{\text{repair}}(G, \phi)$, we also call graph repairs in $\mathcal{S}^{\text{repair}}_{\text{lc}}(G, \phi, U)$ least changing without mentioning the set U for comparison. For example, the graph repair presented in Fig. 7b is least changing.

Finally, we define the notion of delta-preserving graph updates u_2. Such graph updates are constructed by application of the delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ presented in Sect. 8 and which are graph updates that preserve the modification of a previously applied graph update u_1. This means that u_2 does not delete elements that were added by u_1 and that u_2 does not recreate elements that were deleted by u_1. Formally, this means that the composition of u_1 and u_2 is a canonical graph update u.

Definition 18 (**Delta-Preserving Graph Update**)
If $u_1 = (\ell_1, r_1) \in \mathcal{S}^{\text{upd}}$ and $u_2 = (\ell_2, r_2) \in \mathcal{S}^{\text{upd}}$ are graph updates, $u = (\ell, r) \in \mathcal{S}^{\text{upd}}_{\text{can}}$ is a canonical graph update, and $u_1 \circ u_2 = u$, then u_2 *is a delta preserving graph update w.r.t.* u_1, written $u_2 \in \mathcal{S}^{\text{upd}}_{\Delta\text{pres}}(u_1)$.

Other graph repair algorithms (see [28]) attempt to obtain graph repairs that modify the given inconsistent graph by a minimal number of deletions/additions. It turns out that the set of all these *minimal atomic graph repairs* based on a minimal distance is a strict subset of the set of all least changing graph repairs. The statement on the inclusion holds because if a minimal atomic graph repair would not be least changing, then there would be another graph repair with fewer modifications contradicting also the property of being a minimal atomic graph repair. Moreover, the statement on the inclusion being strict holds as demonstrated by the example in Fig. 8. However, as also demonstrated by the example in Fig. 8, the least changing graph repairs provide a more diverse set of graph repairs, which is obtained by our algorithms by incorporating the GC in the construction procedure.

Graph repair algorithms discussed in the remainder of this paper are intended to (a) be sound by only returning graph repairs, (b) be as complete as possible by returning as many least changing graphs repairs as possible, and (c) to always terminate.

We consider two further properties of graph repair algorithms discussed in [28].

Firstly, *stable* graph repair algorithms return the identity update $(\text{id}(G), \text{id}(G))$ when the graph G is already consistent. Obviously, graph repair algorithms for consistency conditions formalized as GCs can easily satisfy this condition by first checking whether the given graph satisfies the GC.

Secondly, *total* graph repair algorithms return at least one repair for every inconsistent graph G, which is a weaker requirement compared to completeness. We consider this property for each of our three graph repair algorithms in the following sections.

6 State-based graph repair

We now introduce two state-based graph repair algorithms for the restoration of consistency, which adhere to the following general interface.

Definition 19 (**State-based Graph Repair Algorithm**)
A state-based graph repair algorithm takes a finite graph $G \in \mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ and a satisfiable consistency constraint $\phi \in$

$\mathcal{S}^{\text{GC}}_{TG,\emptyset}$ as inputs and returns a finite set of graph repairs from $\mathcal{S}^{\text{repair}}(G, \phi)$.

We rely on the tool AUTOGRAPH as discussed in Sect. 4 to determine, using the operation $\mathcal{M}$, the finite set of all minimal graphs satisfying a given GC ϕ. To ease presentation, we assume for the two state-based graph repair algorithms introduced subsequently that the operation $\mathcal{M}$ terminates for all provided inputs and discuss this issue in more detail in subsect. 6.3.

For the demonstration of our algorithms, we make use of a simple running example in which we compute graph repairs for the graph $\mathbf{G}'_{\mathbf{u}}$ from Fig. 5c, which is inconsistent w.r.t. the GC $\boldsymbol{\psi}$ from Fig. 5a.

6.1 State-based repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$

The state-based algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ is designed for the special case that only non-deleting graph repairs are to be constructed. That is, the graph repairs computed by $\mathcal{R}\text{epair}_{\text{sb},1}$ are always of the form $(\text{id}(G), r : G \hookrightarrow G')$ where G is the current graph under repair and where r describes the addition of elements leading to graphs G' satisfying the consistency constraint at hand.

The algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ computes, as a first step, the set $\mathcal{M}(\phi \wedge \exists(\text{i}(G), \top))$ of all minimal graphs that (a) satisfy the consistency constraint given by the GC ϕ and (b) also include a copy of the graph G to be repaired.[2] As a consequence of the construction of this input condition to $\mathcal{M}$, it is guaranteed that every minimal graph G' contained in this set then gives rise to at least one extension monomorphism $r : G \hookrightarrow G'$ from which we obtain one graph repair without deletion.

Definition 20 (Graph Repair Algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$)
If G is a finite typed graph from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ and $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, then $\mathcal{R}\text{epair}_{\text{sb},1}(G, \phi)$ returns the set $\{(\text{id}(G), r : G \hookrightarrow G') \mid G' \in \mathcal{M}(\phi \wedge \exists(\text{i}(G), \top))\}$ of graph repairs.

For our running example ($\boldsymbol{\psi}$ from Fig. 5a and graph $\mathbf{G}'_{\mathbf{u}}$ from Fig. 5c), we do not obtain any graph repair because the loop on node a_2 makes the graph $\mathbf{G}'_{\mathbf{u}}$ inconsistent and any extension of $\mathbf{G}'_{\mathbf{u}}$ also includes this self loop. Hence, there are no non-deleting graph repairs for our running example.

Observe that $\mathcal{R}\text{epair}_{\text{sb},1}$ is stable because we only obtain the non-changing graph repair $(\text{id}(G), \text{id}(G))$ whenever applying $\mathcal{R}\text{epair}_{\text{sb},1}$ to consistent graphs G. Moreover, we compute only least changing graph repairs due to the minimality of the graphs obtained using $\mathcal{M}$ as discussed in Sect. 4 and, vice versa, all graph repairs computed are least changing graph repairs because $\mathcal{M}$ computes the complete set of such minimal graphs.

We state that $\mathcal{R}\text{epair}_{\text{sb},1}$ computes precisely the set of all non-deleting least changing graph repairs.

Theorem 3 (Functional Semantics of $\mathcal{R}\text{epair}_{\text{sb},1}$)
The graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ *is* sound *and* complete *w.r.t. non-deleting least changing graph repairs, upon termination. Formally,* $\mathcal{R}\text{epair}_{\text{sb},1}(G, \phi) = \{(\text{id}(G), r) \mid (\text{id}(G), r) \in \mathcal{S}^{\text{repair}}_{\text{lc}}(G, \phi, \mathcal{S}^{\text{repair}}(G, \phi))\}$.

Note that $\mathcal{R}\text{epair}_{\text{sb},1}$ is not total as it is only complete w.r.t. the non-deleting least changing graph repairs. In fact, the running example demonstrates that $\mathcal{R}\text{epair}_{\text{sb},1}$ is not total already.

6.2 State-based repair algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$

We now introduce our second state-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$, which computes *all* least changing graph repairs. For $\mathcal{R}\text{epair}_{\text{sb},2}$, we refine the approach used for the repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ by computing $\mathcal{M}(\phi \wedge \exists(\text{i}(G_c), \top))$ where suitable inclusion morphisms $\ell : G_c \hookrightarrow G$ describe how G can be restricted to one of its subgraphs G_c. Every graph G' obtained from the application of $\mathcal{M}$for one of these graphs G_c then results in at least one monomorphism $r : G_c \hookrightarrow G'$ resulting in one graph repair returned by $\mathcal{R}\text{epair}_{\text{sb},2}$ (unless it is not a least restrictive graph repair compared to another graph repair computed). That is, ℓ describes the deletion carried out by the resulting graph repair and we apply $\mathcal{M}$ to the graph G_c obtained by the deletion to obtain additions as for the algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$.

We introduce to this extent restriction trees (see Fig. 9 for the restriction tree computed for $\mathbf{G}'_{\mathbf{u}}$ from our running example in simplified notation) that allow to extract such inclusion morphisms ℓ. Given a graph G and a fixed subgraph G_{min} of G, the nodes of the restriction tree are all subgraphs G_c of G that include the graph G_{min}. Note that G_{min} is the empty graph $\emptyset$ in the state-based algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ introduced here but not in the algorithm $\mathcal{R}\text{epair}_{\text{db}}$ introduced later on in Sect. 8. The edges of the restriction tree are given by inclusions that add precisely one node or edge. Obviously, the restriction tree is exponential in $G - G_{min}$, which is problematic when G_{min} is the empty graph $\emptyset$ because the graph G must be assumed to be often not small. Later on in Sect. 8, we use the construction of restriction trees for cases where G is small and $G - G_{min}$ is even smaller. Since the restriction tree is not entirely used in the suboperation $\mathcal{R}\text{epair}_{\text{rec}}$ of $\mathcal{R}\text{epair}_{\text{sb},2}$, we may reduce runtime and memory usage by constructing the restriction tree on-the-fly during an application of $\mathcal{R}\text{epair}_{\text{rec}}$.

Technically, we first construct restrictions trees by first obtaining the set S of all inclusions that are no isomorphisms

[2] We present our recursive algorithms using a mathematical notation as it is more flexible than functional programming and more precise than pseudo code. Moreover, the presented algorithms are constructive, which is demonstrated by our prototypical implementation in AUTOGRAPH.

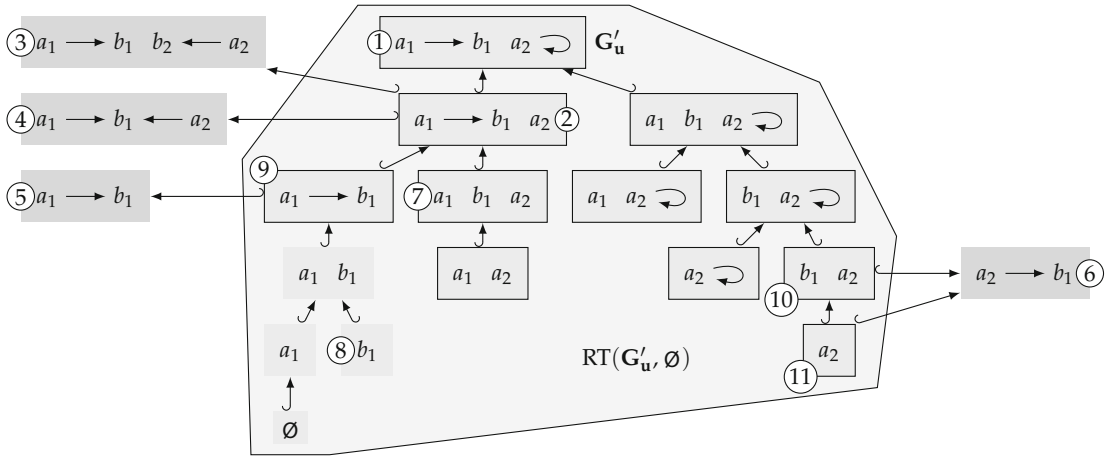


Fig. 9 The restriction tree $\mathrm{RT}(\mathbf{G'_u}, \emptyset)$ (enclosed by the polygon) and four graph repairs (marked 3–6) generated using $\mathcal{R}\mathrm{epair}_{\mathrm{sb},2}$

between two graphs G_c and G_p that are enclosed by G_{min} and G. Then, we obtain the set $S' \subseteq S$ in which all inclusions add precisely one node or edge. Finally, we derive the resulting set $S'' \subseteq S'$ in which we ensure that each graph in the resulting restriction tree is reachable from the root graph G on precisely one path.

Definition 21 (**Restriction Tree**)
If G and G_{min} are finite typed graphs from $\mathcal{S}^{\mathrm{graphs}}_{\mathrm{fin},TG}$, $S = \{\mathrm{inc}(G_c, G_p) \mid G_{min} \subseteq G_c \subset G_p \subseteq G\}$, S' is the least subset of S s.t. the closure of S' under $\circ$ equals S, and S'' is a least subset of S' s.t. when $\ell_1 : G \hookrightarrow G_1 \in S'$ and $\ell_2 : G \hookrightarrow G_2 \in S'$, then at most one of them is in S'', then S'' *is the restriction tree for* G *and* G_{min}, written $\mathrm{RT}(G, G_{min}) = S''$.

While this definition is a rather declarative, we point out that the construction of restriction trees can be implemented easily.

The algorithm $\mathcal{R}\mathrm{epair}_{\mathrm{sb},2}$ is defined using the following operation $\mathcal{R}\mathrm{epair}_{\mathrm{rec}}$ to consider different inclusion morphisms ℓ describing removals of graph elements from the graph to be repaired. In principle, composing all inclusions that constitute one path through the restriction tree from its root describes one viable removal in terms of one such inclusion morphisms ℓ. The operation $\mathcal{R}\mathrm{epair}_{\mathrm{rec}}$ recursively considers for this purpose the graphs in the restriction tree $\mathrm{RT}(G, \emptyset)$ starting with $\mathrm{id}(G)$, denoting the "root" graph G (note that for this initial call to $\mathcal{R}\mathrm{epair}_{\mathrm{rec}}$, the used monomorphism $\mathrm{id}(G)$ is not in the restriction tree). More precisely, $\mathcal{R}\mathrm{epair}_{\mathrm{rec}}$ has four inputs: a graph G to be repaired, a GC ϕ to be satisfied by the repaired graph, an inclusion $\ell : G_c \hookrightarrow G$ that describes an intended removal of graph elements, and a set S of already computed graph repairs using fewer deletions. The recursive traversal computes for the graph G_c, which does not satisfy the GC ϕ, a set of graph repairs by executing $\mathcal{M}(\phi \wedge \exists(\mathrm{i}(G_c), \top))$ as explained above and then descends to the children of G_c to obtain further graph repairs that then include an even more extensive removal upfront. This recursive traversal procedure terminates when the graph G_c already satisfies the GC ϕ, which then leads to the deletion-only graph repair $(\ell : G_c \hookrightarrow G, \mathrm{id}(G_c))$, since smaller graphs would always lead to graph repairs that are not least changing graph repairs in comparison with the graph repair obtained from G_c. Moreover, we ensure that all computed graph repairs are least changing graph repairs by checking that graph repairs computed deeper in the recursive computation are not sub-updates of any of those graph repairs computed already.

Definition 22 (**Repair Operation $\mathcal{R}\mathrm{epair}_{\mathrm{rec}}$**)
If $G \in \mathcal{S}^{\mathrm{graphs}}_{\mathrm{fin},TG}$ is a finite typed graph, $\phi \in \mathcal{S}^{\mathrm{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, $\ell = \mathrm{inc}(G_c, G) : G_c \hookrightarrow G$ is an inclusion morphism, S is a finite set of graph repairs for G w.r.t. ϕ from $\mathcal{S}^{\mathrm{repair}}(G, \phi)$, then $\mathcal{R}\mathrm{epair}_{\mathrm{rec}}(G, \phi, \ell, S) = R$, if one of the following items applies.

- *deletion-only graph repair found*:
  $G_c \models_{\mathrm{GC}} \phi$ *(case of satisfaction)*
  and $R = \{(\ell, \mathrm{id}(G_c))\}$.
- *recursive application*:
  $G_c \not\models_{\mathrm{GC}} \phi$ *(case of non-satisfaction)*,
  $S_1 = \{(\ell, r : G_c \hookrightarrow G') \mid G' \in \mathcal{M}(\phi \wedge \exists(\mathrm{i}(G_c), \top))\}$
  (all graph repairs for current ℓ*)*,
  $S'_1 = \{u_1 \in S_1 \mid \nexists u_2 \in S.\ u_2 \leq u_1\}$ *(retain those without prior computed sub-update)*,
  $S_2 = \bigcup\{\mathcal{R}\mathrm{epair}_{\mathrm{rec}}(G, \phi, \ell \circ \ell', S \cup S'_1) \mid \ell' = \mathrm{inc}(G_d, G_c) \in \mathrm{RT}(G, \emptyset)\}$
  (apply recursively with $S \cup S'_1$ *as found graph repairs)*,
  and $R = S'_1 \cup S_2$
  (return additional graph repairs computed).

The operation $\mathcal{R}\text{epair}_{\text{rec}}$ is guaranteed to terminate because it considers one further graph contained in the finite restriction tree in every recursive application.

For our running example ($\boldsymbol{\psi}$ from Fig. 5a and graph $\mathbf{G}'_{\mathbf{u}}$ from Fig. 5c), we recursively compute the restriction tree depicted in Fig. 9 in simplified notation. We then traverse this restriction tree recursively using $\mathcal{R}\text{epair}_{\text{rec}}$ except for the four graphs without a border such as the graph marked 8, which are not traversed because they have the common supergraph that is marked 9, which satisfies the consistency constraint $\boldsymbol{\psi}$ already. Therefore, traversing those four graphs would generate repairs that are not least changing graph repairs. Hence, the recursive procedure does not reach these graphs and ends in their parents. The resulting graph repairs for our running example are given by the pairs of graphs marked by (2,3), (2,4), (9,5), and (10,6) in Fig. 9. Also note that the graph repair that is given by the two graphs that are marked (11,6) is not a least changing graph repair because of the previously computed graph repair (10,6), which does not delete the b_1 node in between. We therefore do not return this graph repair in the final set $\mathcal{R}\text{epair}_{\text{rec}}(G, \phi, \text{i}(G), \varnothing)$. Another example of such a situation occurs at the graph marked 7 from which the graphs 3 and 4 could also be obtained as extensions: also in this case graph repairs are obtained and then discarded that are not least changing graph repairs.

We now define our second state-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ based on $\mathcal{R}\text{epair}_{\text{rec}}$.

Definition 23 (**Graph Repair Algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$**) If G is a finite typed graph from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ and $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, then $\mathcal{R}\text{epair}_{\text{sb},2}(G, \phi) = \mathcal{R}\text{epair}_{\text{rec}}(G, \phi, \text{id}(G), \varnothing)$ is the set of returned graph repairs.

We state that $\mathcal{R}\text{epair}_{\text{sb},2}$ computes precisely the set of all least changing graph repairs.

Theorem 4 (**Functional Semantics of $\mathcal{R}\text{epair}_{\text{sb},2}$**) *The graph repair algorithm* $\mathcal{R}\text{epair}_{\text{sb},2}$ *is* sound *and* complete *w.r.t. least changing graph repairs, upon termination. Formally,* $\mathcal{R}\text{epair}_{\text{sb},2}(G, \phi) = \mathcal{S}^{\text{repair}}_{\text{lc}}(G, \phi, \mathcal{S}^{\text{repair}}(G, \phi))$.

Note that the totality of the algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ follows immediately from completeness.

6.3 Discussion on state-based repair algorithms

The two state-based graph repair algorithms introduced in this section are independent from the history of the graph to be repaired, which means that no additional information is required for computing the graph repairs. Also, they are able to generate a complete set of all (in the case of $\mathcal{R}\text{epair}_{\text{sb},1}$ deletion-only) least changing graph repairs, which is a stronger property compared to our delta-based graph repair algorithm presented in Sect. 8.

However, the use of AutoGraph for computing $\mathcal{M}$ is costly in these two algorithms especially for cases where the graphs to be repaired are big (an in-depth discussion on the computational complexity of the graph repair algorithms is presented later on in Sect. 9). Moreover, since AutoGraph is not known to terminate for all inputs (cf. Sect. 4), it may happen that the two state-based graph repair algorithms also do not terminate. This is of particular relevance for these algorithms because AutoGraph is used in these two algorithms at runtime on conditions including the graph to be repaired.

Hence, we develop subsequently an incremental delta-based graph repair algorithm for the scenario where a graph is subject to a sequence of updates leading to inconsistent graphs that require the computation of graph repairs after every step. We introduce to this extend an additional data structure in the form of a satisfaction tree (introduced in the next section) to enable incrementality to reduce the computational cost for computing graph repairs when a graph update is provided.

7 Satisfaction trees

We now introduce satisfaction trees (STs), which store information on if and how a graph G satisfies a given GC ϕ (according to Definition 9). We first introduce STs, cover their recursive construction for a given GC ϕ and a graph G, introduce the notion of violations of an ST capturing why the constraint is not satisfied, and finally discuss the propagation of an ST over graph updates to enable its incremental usage in the delta-based graph repair algorithm introduced in the next section.

For the demonstration of satisfaction trees and the delta-based graph repair algorithm from the next section, we extend our running example by also considering the graph update $\mathbf{u}$ given in simplified notation in Fig. 10a that results in the graph $\mathbf{G}'_{\mathbf{u}}$ from Fig. 5c, which is inconsistent w.r.t. the GC $\boldsymbol{\psi}$ from Fig. 5a.

The structure of an ST corresponds to the structure of its corresponding GC, which means that STs are also constructed using the same three operators for conjunction, negation, and existential quantification. In fact, STs can be understood as GCs that are enriched by all of the monomorphisms that could be used during a satisfaction check. In particular, for a given match $m : H \hookrightarrow G$ into the host graph G and a GC $\phi = \exists(f : H \hookrightarrow H', \bar{\phi})$, we store the monomorphisms $m' : H' \hookrightarrow G$ that let the triangle $m = m' \circ f$ commute. Moreover, for each such monomorphism m', we construct and record the ST for m' and the subcondition $\bar{\phi}$.

More precisely, for the case of existential quantification, the corresponding ST is of the forms $\exists(f : H \hookrightarrow H', \bar{\phi}, m_t,$

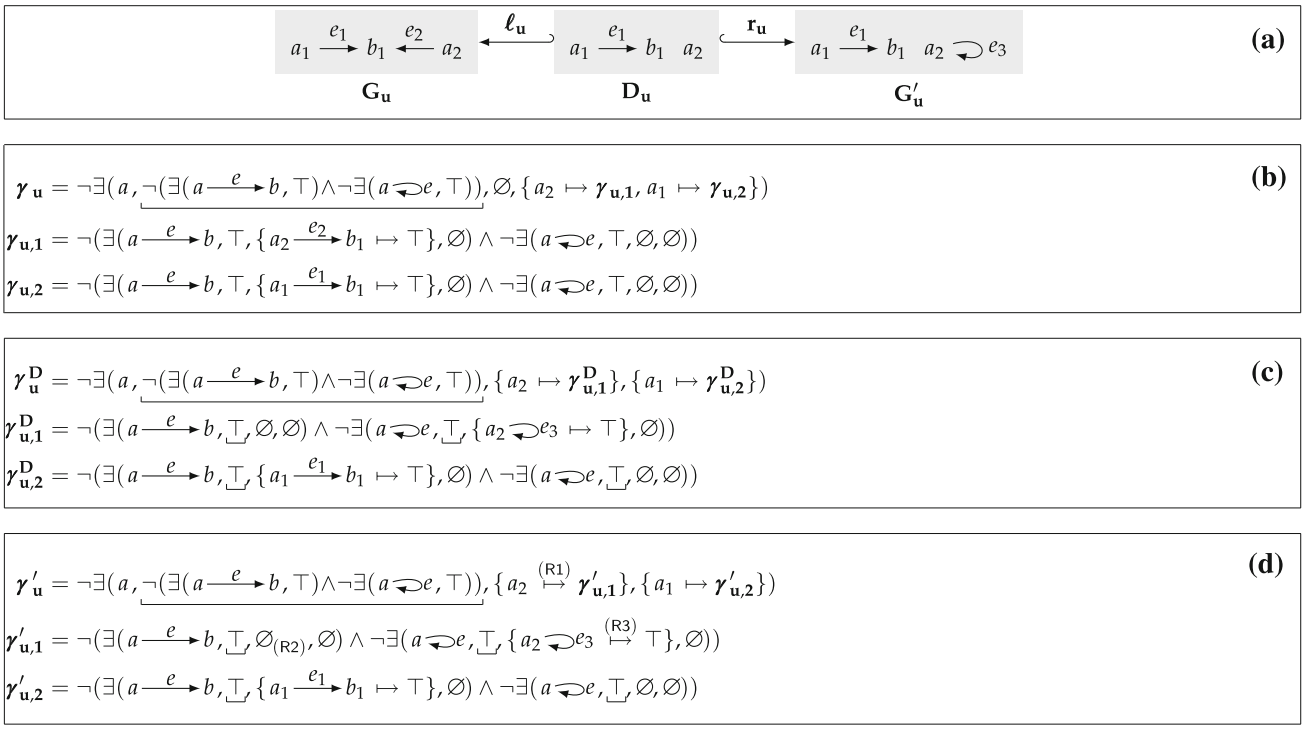


Fig. 10 A graph update and an ST with its propagation over the graph update where GCs are underlined in STs for readability. **a** A graph update $\mathbf{u} = (\boldsymbol{\ell}_\mathbf{u} : \mathbf{D_u} \hookrightarrow \mathbf{G_u}, \mathbf{r_u} : \mathbf{D_u} \hookrightarrow \mathbf{G'_u})$. **b** The ST $\boldsymbol{\gamma}_u$ for $\mathbf{G_u}$ (see **a**) and $\boldsymbol{\psi}$ (see Fig. 5**a**). **c** The ST $\boldsymbol{\gamma}^\mathbf{D}_\mathbf{u}$ for $\mathbf{D_u}$ (see **a**) and $\boldsymbol{\psi}$(see Fig. 5**a**) that is obtained as the backward propagation $\text{ppgB}(\boldsymbol{\gamma}_u, \boldsymbol{\ell}_\mathbf{u})$ from $\boldsymbol{\gamma}_u$ (see **b**) and $\boldsymbol{\ell}_\mathbf{u}$ (see **a**). **d** The ST $\boldsymbol{\gamma}'_\mathbf{u}$ for $\mathbf{G'_u}$ (see **a**) and $\boldsymbol{\psi}$ (see Fig 5a) that is obtained as the forward propagation $\text{ppgF}(\boldsymbol{\gamma}^\mathbf{D}_\mathbf{u}, \mathbf{r_u})$ from $\boldsymbol{\gamma}^\mathbf{D}_\mathbf{u}$ (see **b**) and $\mathbf{r_u}$ (see **a**). Also $\boldsymbol{\gamma}'_\mathbf{u}$ is the result of $\text{ppgU}(\boldsymbol{\gamma}_\mathbf{u}, \mathbf{u})$ that applies backward and forward propagation. The viable points for the delta-based repair discussed in Sect. 8 are indicated by (R1)–(R3)

m_f) where m_t and m_f are partial mappings (we use $\text{support}(g)$ to denote the elements actually mapped by a partial map g) that map matches $m' : H' \hookrightarrow G$ that satisfy $m = m' \circ f$ (for a previously known monomorphism $m : H \hookrightarrow G$) to an ST for the subcondition $\bar{\phi}$. The map m_t maps matches m' to STs for which $m' \models_{\text{GC}} \bar{\phi}$ while m_f maps match m' to STs for which $m' \not\models_{\text{GC}} \phi$.

The following definition describes the syntax of STs. While GCs are defined over their context graph H in Definition 7, we define STs over match morphisms $m : H \hookrightarrow G$ of these context graphs into the given host graph.

Definition 24 (Satisfaction Trees)
If H and G are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, $m : H \hookrightarrow G$ is a monomorphism, then $\gamma \in \mathcal{S}^{\text{GCST}}_{TG,m}$ *is a satisfaction tree (ST) over* m, if one of the following items applies.

- $\gamma = \wedge S$ and $S \subseteq_{\text{fin}} \mathcal{S}^{\text{GCST}}_{TG,m}$.
- $\gamma = \neg\bar{\gamma}$ and $\bar{\gamma} \in \mathcal{S}^{\text{GCST}}_{TG,m}$.
- $\gamma = \exists(f : H \hookrightarrow H', \phi, m_t, m_f), S = \{m' : H' \hookrightarrow G \mid m = m' \circ f\}$ contains the monomorphisms that let the resulting triangle commute, m_t and m_f are finite subsets of $\{(m', \bar{\gamma}) \mid m' \in S \wedge \bar{\gamma} \in \mathcal{S}^{\text{GCST}}_{TG,m'}\}$, and $\phi \in \mathcal{S}^{\text{GC}}_{TG,H'}$ is a GC over H'.

Moreover, we define the following abbreviations.

- true: $\top = \wedge\varnothing$
- false: $\bot = \neg\top$
- disjunction: $\vee S = \neg(\wedge\{\neg\bar{\gamma} \mid \bar{\gamma} \in S\})$
- universal quantification:
  $\forall(f, \phi, m_t, m_f) = \neg\exists(f, \neg\phi, m'_t, m'_f)$
  where $m'_t = \{(m, \neg\bar{\gamma}) \mid (m, \bar{\gamma}) \in m_f\}$
  and $m'_f = \{(m, \neg\bar{\gamma}) \mid (m, \bar{\gamma}) \in m_t\}$

We now define a satisfaction predicate $\models_{\text{ST}}$ for STs for defining when an ST γ defined for a monomorphism $m : H \hookrightarrow G$ states that the contained GC ϕ is satisfied by m.

Definition 25 (Satisfaction of Satisfaction Trees)
If H and G are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, $m : H \hookrightarrow G$ is a monomorphism, and $\gamma \in \mathcal{S}^{\text{GCST}}_{TG,m}$ is a satisfaction tree over m, then $\bar{\gamma}$ *is satisfied*, written $\models_{\text{ST}} \bar{\gamma}$, if one of the following items applies.

- $\gamma = \wedge S$ and $\forall\bar{\gamma} \in S.\ \models_{\text{ST}} \bar{\gamma}$.
- $\gamma = \neg\bar{\gamma}$ and not $\models_{\text{ST}} \bar{\gamma}$.
- $\gamma = \exists(f, \phi, m_t, m_f)$ and $m_t \neq \varnothing$.

Note that the recursive satisfaction predicate does not check the ST underneath an existential quantification as it assumes that the ST is properly constructed. To obtain such properly constructed STs $\bar{\gamma}$, we employ the following recursive operation for a graph G and a condition $\bar{\phi}$ so that $\bar{\gamma}$ represents how

G satisfies (or not satisfies) $\bar{\phi}$. We construct the ST from the STs for the subconditions for the GC operators conjunction and negation. For the case of existential quantification, we obtain all morphisms $m' : H' \hookrightarrow G$ for which the triangle $m = m' \circ f$ commutes and construct the STs for the subcondition ϕ under this extended match m'. The resulting STs are inserted into m_t and m_f according to whether they are satisfied.

Definition 26 (Construction of Satisfaction Trees)
If H and G are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, $m : H \hookrightarrow G$ is a monomorphism, and $\phi \in \mathcal{S}^{\text{GC}}_{TG,H}$ is a graph condition over H, then $\text{cst}(\phi, m) = \gamma$ *is the constructed satisfaction tree for* ϕ *and* m, if one of the following items applies.

- $\phi = \wedge S$ and $\gamma = \wedge\{\text{cst}(\bar{\phi}, m) \mid \bar{\phi} \in S\}$.
- $\phi = \neg\bar{\phi}$ and $\gamma = \neg\text{cst}(\bar{\phi}, m)$.
- $\phi = \exists(f : H \hookrightarrow H', \bar{\phi})$,
  $S = \{m' : H' \hookrightarrow G \mid m = m' \circ f\}$ contains the monomorphisms that let the resulting triangle commute,
  $m_{all} = \{(m', \bar{\gamma}) \mid m' \in S \wedge \text{cst}(\bar{\phi}, m') = \bar{\gamma}\}$ contains the STs constructed for the monomorphisms in S,
  $m_t = \{(m', \bar{\gamma}) \in m_{all} \mid \models_{\text{ST}} \bar{\gamma}\}$ contains the STs that prove satisfaction of ϕ by m, $m_f = m_{all} \setminus m_t$ contains the STs that do not prove satisfaction of ϕ by m, and
  $\gamma = \exists(f, \bar{\phi}, m_t, m_f)$ is the resulting ST.

Also, if $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, then $\text{cst}(\phi, G)$ is equal to the construction of the ST $\text{cst}(\phi, \text{i}(G))$ for the initial monomorphism $\text{i}(G)$.

This recursive construction procedure of STs ensures, for a given graph G and a GC ϕ, that the resulting ST is satisfied if and only if ϕ is satisfied by G. Note that the ST satisfaction relation $\models_{\text{ST}}$ is applied here only on STs that were generated properly using recursive applications of the operation cst.

For our running example, we observe that the ST $\boldsymbol{\gamma_{\mathbf{u}}}$ given in Fig. 10b, which is constructed for the GC $\boldsymbol{\psi}$ from Fig. 5a and the graph $\mathbf{G_u}$ from Fig. 10a, is satisfied.

Theorem 5 (Soundness of the Construction of Satisfaction Trees)
If H and G are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, $m : H \hookrightarrow G$ is a monomorphism, $\phi \in \mathcal{S}^{\text{GC}}_{TG,H}$ is a graph condition over H, and $\text{cst}(\phi, m) = \gamma$ is the constructed ST for ϕ and m, then $\models_{\text{ST}} \gamma$ iff $m \models_{\text{GC}} \phi$.

We now introduce also a detailed handling of the case when an ST γ that is defined for a monomorphism $m : H \hookrightarrow G$ states that the contained GC ϕ is not satisfied by m. This will allow us to reason about the possible points for repairs of a given ST. In particular, we now define a recursive operation (called violations) that determines the set of all violations V contained in an ST. Note that it may be sufficient to repair a single violation to obtain a graph repair that leads to a consistent graph G' from an inconsistent graph G because, intuitively, the operation identifies violations in the form of potential points for repair. Hence, each violation guarantees already on its own that the ST is not satisfied.

The recursive operation violations considers all STs maintained in the ST γ and checks whether such a subcondition is falsely satisfied or falsely not satisfied. The operation uses a Boolean parameter b that is *true* if and only if the current ST is expected to be satisfied. This Boolean parameter is inverted (from *true* to *false* or from *false* to *true*), when the recursion proceeds into a negation.

Before providing the formal definition of the operation violations below, we now discuss the underlying idea in more detail. Note that the cases of conjunction and negation are straightforward as expected and that the cases 3 and 5 from the definition below are not discussed here as they return an empty set of violations for STs that are correctly satisfied or correctly not satisfied.

- A non-empty conjunction $\wedge S$ that is falsely satisfied has only STs in S that are all falsely satisfied. To ensure that any of these STs is no longer satisfied would be a viable repair. Similarly, a non-empty conjunction $\wedge S$ that is falsely not satisfied has at least one ST in S that is falsely not satisfied. To ensure that each of these STs is satisfied would be a viable repair.
- A negation $\neg\bar{\gamma}$ that is falsely satisfied has an ST $\bar{\gamma}$ that is falsely non-satisfied. To ensure that this ST is no longer non-satisfied would be a viable repair. Similarly, a negation $\neg\bar{\gamma}$ that is falsely non-satisfied has an ST $\bar{\gamma}$ that is falsely satisfied. To ensure that this ST is no longer satisfied would be a viable repair. Hence, the value of the Boolean b has to be inverted for the ST $\bar{\gamma}$.
- An ST $\exists(f : H \hookrightarrow H', \phi, m_t, m_f)$ that is falsely non-satisfied has no element in m_t. The resulting violation $(\oplus, f : H \hookrightarrow H', \phi, m : H \hookrightarrow G)$ describes that elements have to be added to the graph (denoted using $\oplus$) to result in an additional match that can be used to satisfy the subcondition ϕ. Also, each match m' mapped by m_f to an ST $\bar{\gamma}$ can't be used to satisfy the subcondition ϕ but graph repairs may affect the STs in $\bar{\gamma}$ resulting in a pair $(m', \bar{\gamma}')$ that would then be inserted into m_t proving the satisfaction of the subcondition ϕ.
- An ST $\exists(f : H \hookrightarrow H', \phi, m_t, m_f)$ that is falsely satisfied has at least one element in m_t. The resulting violation $(\ominus, f : H \hookrightarrow H', \phi, m' : H' \hookrightarrow G)$ describes that elements have to be removed from the graph (denoted using $\ominus$) to result in all these matches to be invalidated by removing elements matched by such a monomorphism m'. Also, each match m' mapped by m_t to an ST $\bar{\gamma}$ can be used to satisfy the subcondition ϕ but graph repairs may affect the STs in $\bar{\gamma}$ possibly resulting in a pair $(m', \bar{\gamma}')$ that would then be inserted into m_f for not proving the satisfaction of the subcondition ϕ anymore.

We now provide the definition of the operation for obtaining violations from an ST.

Definition 27 (Violations of a Satisfaction Tree) If H and G are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, $m : H \hookrightarrow G$ is a monomorphism, $\gamma \in \mathcal{S}^{\text{GCST}}_{TG,m}$ is an ST over m, and $b \in \mathbf{B}$ is a Boolean value, then violations$(\gamma, b) = V$ *is the set of violations of* γ *for* b, if one of the following items applies.

- $\gamma = \wedge S$ and $V = \bigcup\{\text{violations}(\bar{\gamma}, b) \mid \bar{\gamma} \in S\}$.
- $\gamma = \neg\bar{\gamma}$ and $V = \text{violations}(\bar{\gamma}, \neg b)$.
- $\gamma = \exists(f, \phi, m_t, m_f)$, $b = true$, $m_t \neq \varnothing$, and $V = \varnothing$.
- $\gamma = \exists(f, \phi, m_t, m_f)$, $b = true$, $m_t = \varnothing$, and $V = \{(\oplus, f, \phi, m)\} \cup \bigcup\{\text{violations}(\bar{\gamma}, b) \mid (m', \bar{\gamma}) \in m_f\}$.
- $\gamma = \exists(f, \phi, m_t, m_f)$, $b = false$, $m_t = \varnothing$, and $V = \varnothing$.
- $\gamma = \exists(f, \phi, m_t, m_f)$, $b = false$, $m_t \neq \varnothing$, and $V = \bigcup\{(\ominus, f, \phi, m) \mid (m, \bar{\gamma}) \in m_t\} \cup \bigcup\{\text{violations}(\bar{\gamma}, b) \mid (m', \bar{\gamma}) \in m_t\}$.

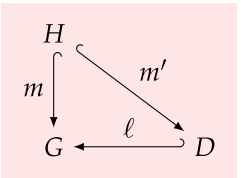
We state that the set of violations derived using this operation from an ST is compatible with the satisfaction predicate $\models_{\text{ST}}$ from Definition 25. This means that an ST is satisfied if and only if no violation can be obtained from it.

Theorem 6 (Compatibility of Satisfaction of Satisfaction Trees and Computation of Violations) *If H and G are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, $m{:}H \hookrightarrow G$ is a monomorphism, $\gamma \in \mathcal{S}^{\text{GCST}}_{TG,m}$ is an ST over m, then $\models_{\text{ST}} \gamma$ iff* violations$(\gamma, true) = \varnothing$.

Subsequently, we define the operation ppgU for the propagation of a given ST γ that is constructed for a graph G over a graph update $(\ell : D \hookrightarrow G, r : D \hookrightarrow G')$ to obtain an ST γ' such that $\gamma' = \text{cst}\ (\phi, G')$ whenever $\gamma = \text{cst}\ (\phi, G)$. That is, the propagation of the ST results in the same ST that would have been constructed directly using the operation cst from above. This update propagation over an update using the operation ppgU is performed in two steps. The first step is a backward propagation of γ for $\ell : D \hookrightarrow G$ using the operation ppgB (defined later in Definition 29) and the second step is a forward propagation of the resulting ST for $r : D \hookrightarrow G'$ using the operation ppgF (defined later in Definition 30).

For backward propagation, we describe how the deletion of elements in G by $\ell : D \hookrightarrow G$ affects its associated ST γ. To this end, we first explain how matches $m : H \hookrightarrow G$ occurring in an ST are propagated over $\ell : D \hookrightarrow G$. The outcome of this match propagation is a monomorphism $m' : H \hookrightarrow D$ satisfying $\ell \circ m' = m$. That is, m' is the restriction of m to D, which exists uniquely when every element matched by m is also matched (i.e., preserved) by ℓ (formally, $m(G) \subseteq \ell(D)$). Matches $m : H \hookrightarrow G$ where some elements matched by m are deleted by ℓ cannot be preserved by the propagation; the operation ppgMatch is therefore a partial map returning the undefined element $\perp$ in this case.

Definition 28 (Propagation of Match) If H, G, and D are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ and $m : H \hookrightarrow G$ and $\ell : D \hookrightarrow G$ are monomorphisms, then ppgMatch$(m, \ell) = m' : H \hookrightarrow D$ *is the propagation of m over ℓ*, if m' satisfies $\ell \circ m' = m$ if such a monomorphism m' exists. Otherwise, if such a monomorphism $m' : H \hookrightarrow D$ does not exist, then we define ppgMatch$(m, \ell) = \perp$ to return the "undefined element" $\perp$.



The following recursive operation for the backward propagation defines how deletions given by a monomorphism $\ell : D \hookrightarrow G$ affect the maps m_t and m_f of the given ST. That is, when $\bar{\gamma} = \exists(f, \phi, m_t, m_f)$ and (m, γ) is a mapping contained in m_t or m_f, we have two cases. If the match m can not be propagated (i.e., ppgMatch$(m, \ell) = \perp$), we remove the mapping. Alternatively, if the match m can be propagated to a match m' (i.e., ppgMatch$(m, \ell) = m' \neq \perp$), we construct the mapping pair $(m', \text{ppgB}(\ell, \gamma))$ and check whether this updated pair belongs to the resulting map m'_t or m'_f of the resulting ST. Note that matches that were used to show that the subcondition was (or was not) satisfied may be matches that can be used to show that the subcondition is not (or is) satisfied.

Definition 29 (Backward Propagation) If H, G, and D are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ $m : H \hookrightarrow G$ is a monomorphism, $\gamma \in \mathcal{S}^{\text{GCST}}_{TG,m}$ is an ST over m, $\ell : D \hookrightarrow G$ is a monomorphism describing a deletion, ppgMatch$(m, \ell) = m' : H \hookrightarrow D$ is the propagation of m over ℓ, and $\bar{\gamma} \in \mathcal{S}^{\text{GCST}}_{TG,m'}$ is an ST over m', then ppgB$(\gamma, \ell) = \bar{\gamma}$ *is the backward propagation of γ over ℓ*, if one of the following items applies.

- $\gamma = \wedge S$ and $\bar{\gamma} = \wedge\{\text{ppgB}(\gamma', \ell) \mid \gamma' \in S\}$.
- $\gamma = \neg\gamma'$ and $\bar{\gamma} = \neg\text{ppgB}(\gamma', \ell)$.
- $\gamma = \exists(f : H \hookrightarrow H', \phi, m_t, m_f)$, $m_{all} = \{(m', \text{ppgB}(\gamma', \ell)) \mid (m, \gamma') \in m_t \cup m_f \wedge \text{ppgMatch}(m, \ell) = m' \neq \perp\}$, $m'_t = \{(m', \bar{\gamma}') \in m_{all} \mid \models_{\text{ST}} \bar{\gamma}'\}$, $m'_f = m_{all} \setminus m'_t$, and $\bar{\gamma} = \exists(f, \phi, m'_t, m'_f)$.

Note that the initial monomorphism $\text{i}(G) : \emptyset \hookrightarrow G$ can be propagated over any deletion monomorphism $\ell : D \hookrightarrow G$

resulting in the monomorphism $\mathrm{i}(D)$, and, hence, the operation ppgB is applicable to all STs $\gamma \in \mathcal{S}^{\mathrm{GCST}}_{TG,\mathrm{i}(G)}$, which is sufficient as we define consistency constraints using GCs only over the empty graph and hence obtain STs contained in $\mathcal{S}^{\mathrm{GCST}}_{TG,\mathrm{i}(G)}$ later on.

For our running example, we construct the ST $\boldsymbol{\gamma}^{\mathbf{D}}_{\mathbf{u}}$ given in Fig. 10c using backward propagation of the ST $\boldsymbol{\gamma}_{\mathbf{u}}$ over the monomorphism $\boldsymbol{\ell}_{\mathbf{u}}$ of the considered graph update.

For soundness of the operation ppgB, we state that the ST obtained using ppgB equals the one that would be obtained when constructing the ST from scratch using the operation cst from before.

Lemma 1 (Compatibility of Satisfaction Tree Construction and Backward Propagation)
If G and D are finite typed graphs from $\mathcal{S}^{\mathrm{graphs}}_{\mathrm{fin},TG}$, $\ell : D \hookrightarrow G$ is a monomorphism describing a deletion, and $\phi \in \mathcal{S}^{\mathrm{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph,
then $\mathrm{ppgB}(\mathrm{cst}(\phi, G), \ell) = \mathrm{cst}(\phi, D)$.

For the second step of propagation, we consider now a monomorphism $r : D \hookrightarrow G'$ and apply the subsequently defined forward propagation on the ST constructed for D to obtain an ST constructed for G'. In this case of forward propagation, where additions are given by $r : D \hookrightarrow G'$, we can preserve all matches $m : H \hookrightarrow D$ resulting in monomorphisms $r \circ m : H \hookrightarrow G'$. However, as for the backward propagation, we note that the addition of further elements specified by $r : D \hookrightarrow G$ can affect the satisfaction of the propagated match $r \circ m$, which requires again that the resulting ST is checked for satisfaction in each case to ensure that the adapted mappings are inserted into the right partial map m'_t and m'_f. Also, the addition of elements can result in matches that were not available before; for these additional matches, we must construct STs from scratch using the operation cst.

Definition 30 (Forward Propagation)
If H, D, and G' are finite typed graphs from $\mathcal{S}^{\mathrm{graphs}}_{\mathrm{fin},TG}$, $m : H \hookrightarrow D$ is a monomorphism, $\gamma \in \mathcal{S}^{\mathrm{GCST}}_{TG,m}$ is an ST over m, $r : D \hookrightarrow G'$ is a monomorphism describing an addition, and $\bar{\gamma} \in \mathcal{S}^{\mathrm{GCST}}_{TG,r\circ m}$ is an ST over $r \circ m$, then $\mathrm{ppgF}(\gamma, r) = \bar{\gamma}$ *is the forward propagation of γ over r*, if one of the following items applies.

- $\gamma = \wedge S$ and $\bar{\gamma} = \wedge\{\mathrm{ppgF}(\gamma', r) \mid \gamma' \in S\}$.
- $\gamma = \neg\gamma'$ and $\bar{\gamma} = \neg\mathrm{ppgF}(\gamma', r)$.
- $\gamma = \exists(f : H \hookrightarrow H', \phi, m_t, m_f)$,

$$
\begin{aligned}
m_{adapted} &= \{(r \circ m, \mathrm{ppgF}(\gamma', r)) \mid (m, \gamma') \in m_t \cup m_f\}\\
m_{new} &= \{(m', \mathrm{cst}(\phi, m')) \mid \\
&\qquad r \circ m = m' \circ f \wedge \\
&\qquad m' \notin \mathrm{support}(m_{adapted})\},\\
m_{all} &= m_{adapted} \cup m_{new},
\end{aligned}
$$

$m'_t = \{(m', \bar{\gamma}') \in m_{all} \mid \models_{\mathrm{ST}} \bar{\gamma}'\}$, $m'_f = m_{all} \setminus m'_t$,
and $\bar{\gamma} = \exists(f, \phi, m'_t, m'_f)$.

For our running example, we derive the ST $\boldsymbol{\gamma}'_{\mathbf{u}}$ given in Fig. 10d using forward propagation of the ST $\boldsymbol{\gamma}^{\mathbf{D}}_{\mathbf{u}}$ over the monomorphism $\mathbf{r}_{\mathbf{u}}$ of the considered graph update.

As for the operation ppgB, we state that the operation ppgF incrementally computes the ST that would be obtained when constructing the ST for the target graph G' from scratch using the operation cst.

Lemma 2 (Compatibility of Satisfaction Tree Construction and Forward Propagation)
If D and G' are finite typed graphs from $\mathcal{S}^{\mathrm{graphs}}_{\mathrm{fin},TG}$, $r : D \hookrightarrow G'$ is a monomorphism describing an addition, and $\phi \in \mathcal{S}^{\mathrm{GC}}_{TG,\emptyset}$ is a GC over the empty graph,
then $\mathrm{ppgF}(\mathrm{cst}(\phi, G), r) = \mathrm{cst}(\phi, G')$.

To obtain the propagation operation ppgU that propagates a given ST constructed for a graph G over a graph update $(\ell : D \hookrightarrow G, r : D \hookrightarrow G')$, which modifies G into G', we now compose the operation for backward propagation and the operation for forward propagation.

Definition 31 (Update Propagation)
If H is a finite typed graph from $\mathcal{S}^{\mathrm{graphs}}_{\mathrm{fin},TG}$, $u = (\ell : D \hookrightarrow G, r : D \hookrightarrow G') \in \mathcal{S}^{\mathrm{upd}}$ is a graph update, $m : H \hookrightarrow G$ is a monomorphism, $\gamma \in \mathcal{S}^{\mathrm{GCST}}_{TG,m}$ is an ST over m, and $\mathrm{ppgMatch}(m, \ell) = m'$ is the propagation of m over ℓ, then $\mathrm{ppgU}(\gamma, u) = \mathrm{ppgF}(\mathrm{ppgB}(\gamma, \ell), r)$ *is the propagation of γ over u*, which is an ST over $r \circ m'$ from $\mathcal{S}^{\mathrm{GCST}}_{TG,r\circ m'}$.

Finally, we state that the operation ppgU returns the ST that would be obtained when constructing the ST from scratch using the operation cst.

Theorem 7 (Compatibility of Satisfaction Tree Construction and Update Propagation)
If $u = (\ell : D \hookrightarrow G, r : D \hookrightarrow G') \in \mathcal{S}^{\mathrm{upd}}$ is a graph update and $\phi \in \mathcal{S}^{\mathrm{GC}}_{TG,\emptyset}$ is a GC over the empty graph,
then $\mathrm{ppgU}(\mathrm{cst}(\phi, G), u) = \mathrm{cst}(\phi, G')$.

Note that finding matches $m : H \hookrightarrow G$ into a given graph G according to the satisfaction relation of GCs is NP-complete but the development of static and dynamic heuristics for matching graphs is an active field of research [3,6,8,11,18,23]. However, note that the efficiency of matching algorithms depends primarily on the size of the host graph G since the subgraph isomorphism problem has polynomial

complexity for a fixed pattern H [45,46] and because the graph pattern H can be assumed to be small compared to the host graph G. Still, we consider the overall propagation given by ppgU to be *incremental* in the sense that the operation cstis only used in the forward propagation on parts of the graph G' where the addition of graph elements via r results in additional matches m'. These additional matches must then map to at least one element that was added by the monomorphism r. The time that is required for deriving all such additional matches m' can be greatly reduced when all elements in the graphs to be matched are connected. The resulting search for matches is then local to the addition and therefore more efficient in general. However, this connectedness condition is not satisfied by consistency constraints given by GCs in general. Also, as discussed later on in more detail, the addition of a single graph element may result in a single match, which then triggers the construction of an exponential number of additional STs (as demonstrated in Fig. 17).

Based on the STs introduced in this section, we introduce our delta-based graph repair algorithm in the next section, which determines graph repairs from ST that are not satisfied.

8 Delta-based graph repair

We now introduce a delta-based graph repair algorithm for the restoration of consistency, which adheres to the following general interface.

Definition 32 (Delta-based Graph Repair Algorithm) A delta-based graph repair algorithm takes a finite graph $G \in \mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, a graph update $u = (\ell : D \hookrightarrow G, r : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$, a satisfiable consistency constraint $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$, and a finite state q as inputs and returns a finite set of pairs (u', q') of a graph repair $u' \in \mathcal{S}^{\text{repair}}(G, \phi)$ and a finite state q'.

In contrast to the two state-based graph repair algorithms, we permit that delta-based graph repair algorithms make use of a storage recording a finite state q to maintain knowledge about the graph that is monitored. In our delta-based graph repair algorithm, this finite state value $q = (\gamma, M)$ is given by (a) the ST γ that is equal to the ST that would be constructed for the current graph G and the user-provided consistency constraint ϕ and (b) an offline constructed map M that assigns to each subcondition $\phi' \in \mathcal{S}^{\text{GC}}_{TG,H}$ of ϕ (i.e., $\phi' \in \text{sub}(\phi)$) the finite set of minimal graphs satisfying ϕ' as computed using $\mathcal{M}(\exists(\text{i}(H), \phi'))$ according to Definition 10. The ST is propagated at runtime over the externally controlled graph updates, which may result in inconsistency, as well as over the graph repairs computed by our delta-based graph repair algorithm. While this maintenance of the ST imposes additional costs, it also greatly reduces the time required to determine violations of consistency for an adapted graph. The map M is not modified at runtime but used for the computation of graph repairs as discussed in detail later on.

The procedure for obtaining violations as given in Sect. 7 and our discussion before Definition 27 already indicate how additions and removals of graph elements can be used to repair an inconsistent graph by repairing its violations. In particular, our delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ has the inputs of a finite graph G, a graph update $u = (\ell : D \hookrightarrow G, r : D \hookrightarrow G')$, and a satisfiable consistency constraint given by a GC $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ and uses the ST $\gamma = \text{cst}(\phi, G)$ in its state variable (γ, M). Firstly, it propagates the ST γ using the operation ppgU for the provided graph update u to obtain the ST $\gamma' = \text{cst}(\phi, G')$ used in the updated state variable (γ', M). Secondly, it computes the set of all violations from the ST γ'. Thirdly, if necessary, it employs the single-step graph repair algorithm $\mathcal{R}\text{epair}_{\text{db1}}$ to obtain a repair for a violation. That is, $\mathcal{R}\text{epair}_{\text{db1}}$ handles only a single violation of the graph G' and thereby operates at the local level determined by the considered violation. A consequence of this local repair approach is that a graph update derived for a single violation does not repair the entire graph in general because it may be necessary that multiple violations require a treatment in the final graph repair to be computed. Hence, we employ $\mathcal{R}\text{epair}_{\text{db1}}$ iteratively to obtain a sequence of graph updates for repairing a sequence of violations until a consistent graph is obtained; in this case, we define the composition of the computed graph updates to be the final graph repair.

However, the repair of a single violation may result in a graph with more, less, or the same number of violations in general. See Fig. 12 for an example where the number of violations rises with graph updates computed for violations before the number of violations is successfully reduced to zero. Hence, there is no guarantee that the iterative computation of repairs for violations terminates in a consistent graph. For this reason, we employ $\mathcal{R}\text{epair}_{\text{db1}}$ in $\mathcal{R}\text{epair}_{\text{db}}$ in breadth-first manner to ensure that every graph repair that can be obtained using this multi-step approach is indeed obtained eventually. That is, using breadth-first search ensures that we gradually compute the desired set of graph repairs.[3]

For our running example from Fig. 10a, such a multi-step repair of $\mathbf{G}'_{\mathbf{u}}$ is given in Fig. 13 where the obtained graph updates result in the graphs marked 1–3, of which only the graph marked 1 already satisfies the consistency constraint $\boldsymbol{\psi}$. The delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ then continues to apply $\mathcal{R}\text{epair}_{\text{db1}}$ for the inconsistent graphs marked 2–3 to compute further graph updates resulting in the graphs marked 1 and 4 where the graph marked 4 also satisfies $\boldsymbol{\psi}$. The graph $\boldsymbol{\gamma}'_{\mathbf{u}}$ has two violations: on the one hand, there is

[3] In Sect. 9, we also discuss the impact on runtime when using depth-first search instead for the special case when only a single graph repair is to be obtained.

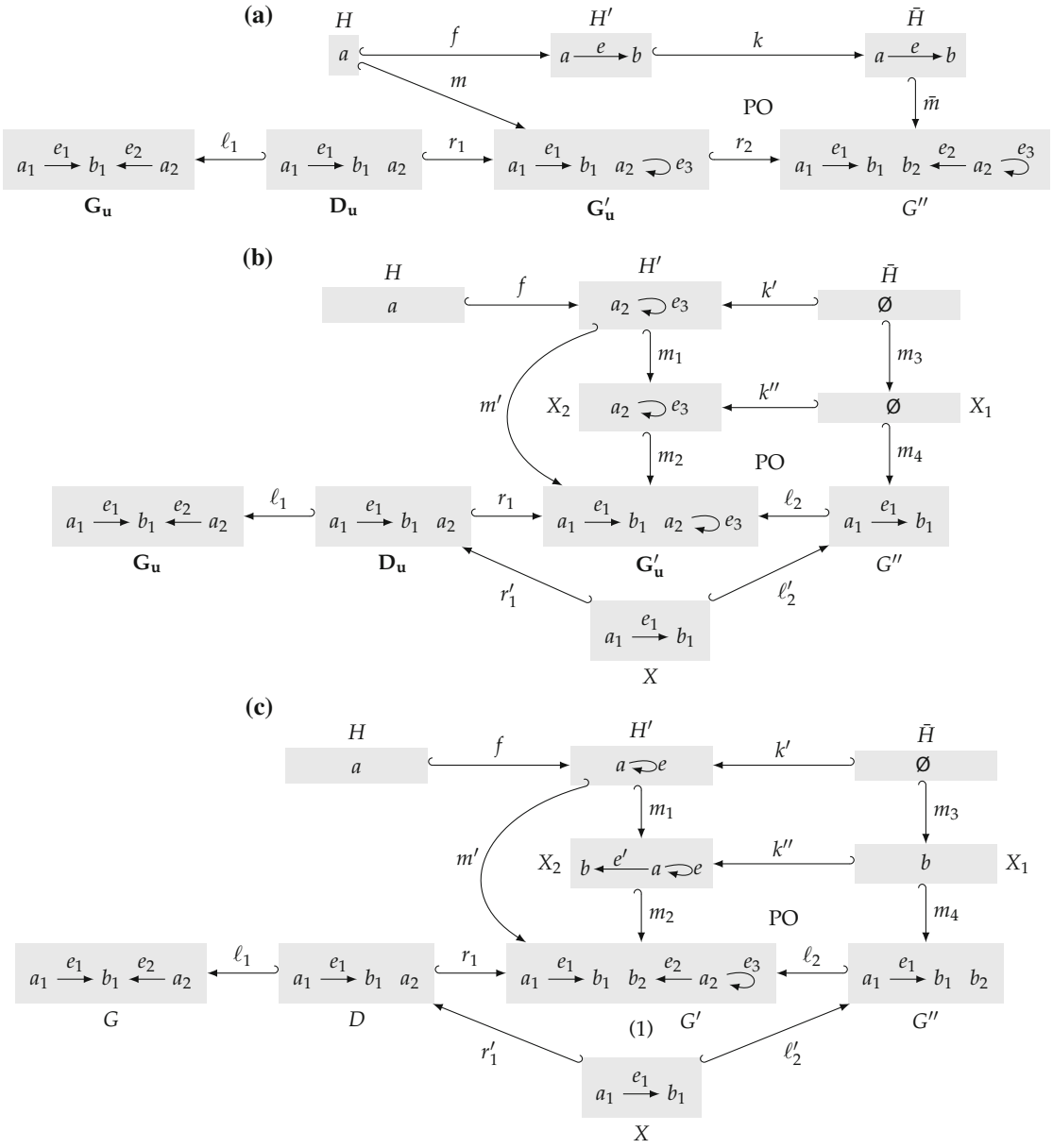


Fig. 11 Applications of local graph repair operations $\mathcal{R}\text{epair}_{\text{add}}$ and $\mathcal{R}\text{epair}_{\text{del}}$. **a** An application of the local addition-based graph repair according to Definition 34. The repair step R2 (see the marking in Fig. 10d) results in the graph marked 2 in Fig. 13. "It adds the node b_2 and an edge e_2 from a_2 to b_2 to establish a graph G'' satisfying $\boldsymbol{\psi}$. The repair is necessary because for the match m, there is no consistent extension with respect to the monomorphism f. Then AUTOGRAPH is used to create the monomorphism k that leads to a graph $\bar{H}$ (H' and $\bar{H}$ are identical here because the subcondition of the considered GC $\exists(f, \top)$ is $\top$ not requiring further graph elements). Finally, to integrate these additional elements into the current graph $\mathbf{G}'_{\mathbf{u}}$, we construct the pushout. **b** An application of the local deletion-based graph repair according to Definition 35. The repair step R1 (see the marking in Fig. 10d) results in the graph marked 1 in Fig. 13. It removes the node a_2 with the loop to obtain the graph G'' satisfying $\boldsymbol{\psi}$. The ST $\boldsymbol{\gamma}'_{\mathbf{u}}$ contains the match m' that is consistent with the previous match m and the morphism f from the existential quantification. Note that the additional construction of X_2 and X_1 is not required here because the node a_2 has no further edges attached that would need to be deleted in addition. **c** Another application of the local deletion-based graph repair according to Definition 35. The given graph update is obtained as the composition of the graph update **u** from Fig. 10a and the graph update leading to the graph marked 2 in Fig. 13. The graph G' is inconsistent because of the loop on the node a_2. The presented diagram describes a derived local deletion-based graph repair that removes the node a_2 with the two attached edges. Note that this local graph repair is not delta-preserving because the pullback (1) is not a pushout (the local graph repair removes in ℓ_2 the edges that were added in r_1)

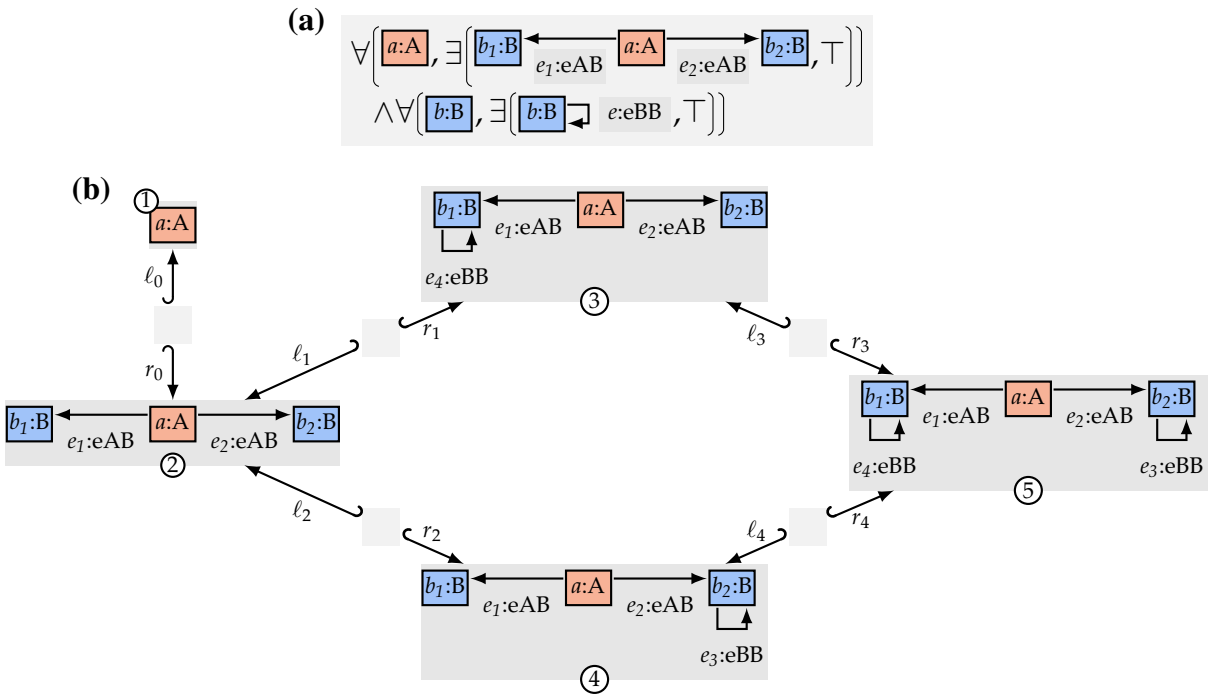


Fig. 12 An example of an iterated computation of local graph repairs. **a** A GC representing a consistency constraint stating that every :A node should have two connected :B nodes and that every :B node has a self-loop. **b** A successful local graph repair computation that increases the number of violations before reducing the number of violations to zero afterwards. Given the graph marked 1 with a single violation, we obtain a unique local repair that adds two :B nodes resulting in the graph marked 2. Given the graph marked 2 with two violations, we obtain two unique local repairs that add self-loops to each of the two :B nodes resulting in the graph marked 3 and 4. Finally, each of the two graphs marked 3 and 4 is then repaired by adding the missing self-loop resulting in the same graph marked 5

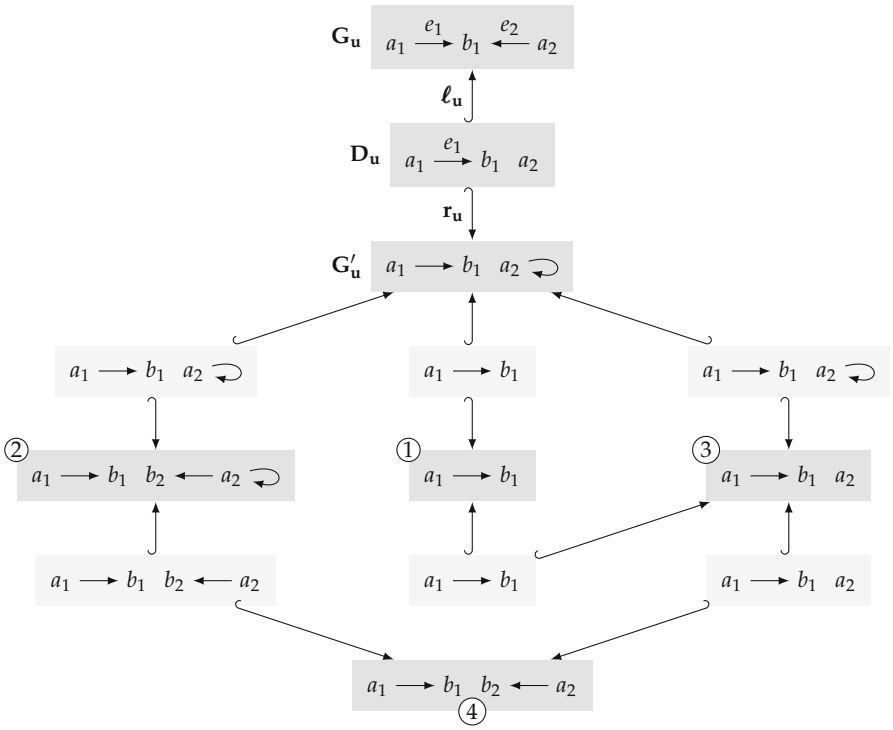


Fig. 13 An example for delta-based graph repair using $\mathcal{R}\text{epair}_{\text{db}}$

a missing :B node that must be connected to a_2 and, on the other hand, there is a forbidden self-loop on the a_2 node. The graphs marked 1, 2, 3, and 4 have zero, one (again, the forbidding self-loop on a_2), one (again, the missing connected :B node), and zero violations, respectively.

We now first introduce least changing local graph repairs u for a graph G_1 with a violation v, which we expect to be computed by $\mathcal{R}\text{epair}_{\text{dbl}}$, as the graph updates that remove the corresponding violation v' from a minimal context graph G'_1 that is contained in G_1 such that the graph update u' performed on G'_1 can be embedded into G_1 resulting in the graph update u via a double pushout diagram such that the same violation is repaired for G_1 (see [15,37] for a thorough introduction to the DPO approach to typed graph transformation). For this purpose, we distinguish between local repairs using addition and local repairs using deletion. In both cases, we ensure that the local repair that modifies the minimal context G'_1 into a resulting graph G'_2 also translates to the embedding where the same local repair is executed due to the DPO step and where we require in addition that the translated repair also succeeds in removing the violation at hand.

Definition 33 (**Least Changing Local Graph Repairs**) If $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph and $u = (\ell : D \hookrightarrow G_1, r : D \hookrightarrow G_2) \in \mathcal{S}^{\text{upd}}$ is a graph update, then *u is a least changing local graph repair of G w.r.t. ϕ*, written $u \in \mathcal{S}^{\text{repair}}_{\text{lcl}}(G, \phi)$, if there is a minimal restriction of G_1 given by a monomorphism $e_1 : G'_1 \hookrightarrow G_1$ s.t.

- $u' = (\ell' : D' \hookrightarrow G'_1, r' : D' \hookrightarrow G'_2) \in \mathcal{S}^{\text{upd}}$ is a graph update,
- $\gamma = \text{cst}(\phi, G_1)$ is the ST constructed for ϕ and G_1,
- $\gamma' = \text{cst}(\phi, G'_1)$ is the ST constructed for ϕ and G'_1,
- $v \in \text{violations}(\gamma, \mathit{true})$ is a violation of γ,
- $v' \in \text{violations}(\gamma', \mathit{true})$ is a violation of γ',
- the squares in the diagrams below are pushouts,

and one of the following items applies.

- local graph repair by addition (see Fig. 14a for a visualization):
  - $v = (\oplus, f : H_1 \hookrightarrow H_2, \bar{\phi}, m_1 : H_1 \hookrightarrow G_1)$ is a violation requiring an addition,
  - $v' = (\oplus, f : H_1 \hookrightarrow H_2, \bar{\phi}, m'_1 : H_1 \hookrightarrow G'_1)$ is a violation requiring an addition,
  - $m_1 = e_1 \circ m'_1$,
  - $\text{ppgMatch}(m'_1, \ell') = m'_2 : H_1 \hookrightarrow D'$,
  - $r' \circ m'_2 \models_{\text{GC}} \exists(f, \bar{\phi})$,
  - $\text{ppgMatch}(m_1, \ell) = m_2 : H_1 \hookrightarrow D$, and
  - $r \circ m_2 \models_{\text{GC}} \exists(f, \bar{\phi})$.
- local graph repair by deletion (see Fig. 14b for a visualization):
  - $v = (\ominus, f : H_1 \hookrightarrow H_2, \bar{\phi}, m_1 : H_2 \hookrightarrow G_1)$ is a violation requiring a deletion,

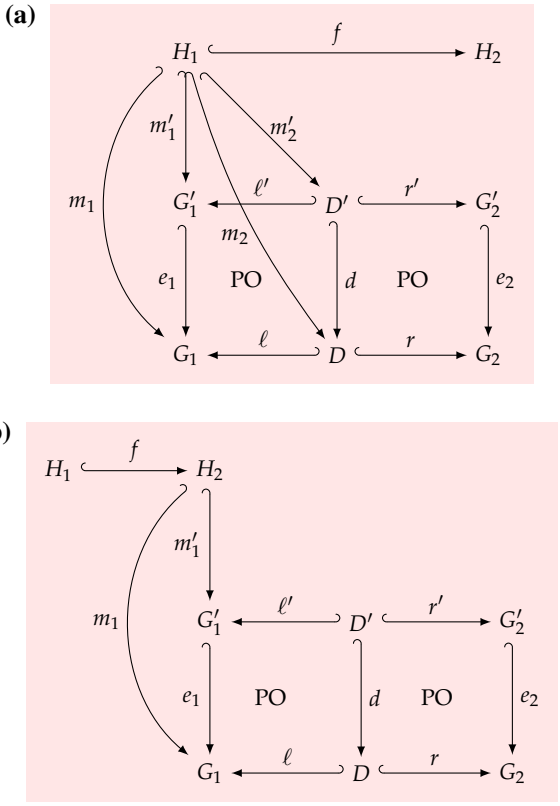


Fig. 14 Visualization for Definition 33. **a** Visualization for local graph repair by addition. **b** Visualization for local graph repair by deletion

  - $v' = (\ominus, f : H_1 \hookrightarrow H_2, \bar{\phi}, m'_1 : H_2 \hookrightarrow G'_1)$ is a violation requiring a deletion,
  - $m_1 = e_1 \circ m'_1$, and
  - $\text{ppgMatch}(m'_1, \ell') = \bot$.

Now we describe how to obtain such least changing local graph repairs for violations by addition or deletion. The operation $\mathcal{R}\text{epair}_{\text{dbl}}$ therefore depends on two local graph repair operations $\mathcal{R}\text{epair}_{\text{add}}$ and $\mathcal{R}\text{epair}_{\text{del}}$ for deriving single-step repairs that add and delete elements from the graph under repair.

For $\mathcal{R}\text{epair}_{\text{add}}$, a GC $\exists(f : H \hookrightarrow H', \phi')$ occurring as a subcondition in the consistency constraint ϕ may be violated because, for the match $m : H \hookrightarrow G'$, which locates a copy of H in the graph G' under repair, no suitable match $m' : H' \hookrightarrow G'$ can be found for which $m = m' \circ f$ and $m' \models_{\text{GC}} \phi'$ are satisfied. The local graph repair operation $\mathcal{R}\text{epair}_{\text{add}}$ resolves this violation by (a) using the map M generated using AUTOGRAPH to select a suitable graph $\bar{H}$, (b) integrating this graph $\bar{H}$ into G' resulting in G'' such that a suitable match $m' : H' \hookrightarrow G''$ can be found (where $m' = \bar{m} \circ k \circ f$ in the following definition), and (c) checking whether the monomorphism $r_2 : G' \hookrightarrow G''$ adds elements that were removed in the provided graph update

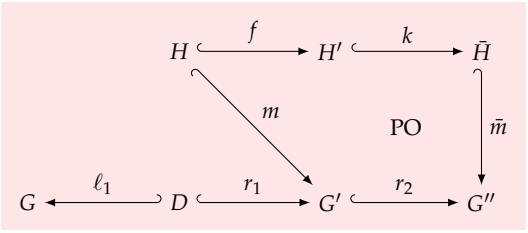


Fig. 15 Visualization for Definition 34

$u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ using the monomorphism ℓ_1 by checking whether the composition of u with the computed graph update is canonical.

Definition 34 (Addition-based Local Graph Repair)
If (see Fig. 15 for a visualization)

- $u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ is a graph update,
- $f : H \hookrightarrow H'$ is a monomorphism,
- $\phi \in \mathcal{S}^{\text{GC}}_{TG,H'}$ is a GC defined over H',
- $m : H \hookrightarrow G'$ is a monomorphism,
- $\bar{H} \in M(\exists(f, \phi))$ is an addition recorded by M,
- $k : H' \hookrightarrow \bar{H}$ is a monomorphism describing the addition to $\bar{H}$,
- $(\bar{m}, r_2)$ is the pushout of $(m, k \circ f)$,
- $b \in \mathbf{B}$ states whether the graph update is a canonical graph update,
- $b = \mathit{true}$ iff $(\ell_1, r_2 \circ r_1) \in \mathcal{S}^{\text{upd}}_{\text{can}}$, and
- $\bar{m} \circ k \models_{\text{GC}} \phi$ states that the addition results in a locally satisfied GC ϕ,

then $((\text{id}(G'), r_2), b) \in \mathcal{R}\text{epair}_{\text{add}}(u, f, \phi, m)$.

Note that the Boolean value b is used to check whether the graph update obtained by $\mathcal{R}\text{epair}_{\text{add}}$ is delta preserving w.r.t. the provided graph update u.

Lemma 3 (Addition-based Local Graph Repair Results in Delta Preserving Graph Updates)
If $u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$, $f : H \hookrightarrow H'$ is a monomorphism, $\phi \in \mathcal{S}^{\text{GC}}_{TG,H'}$ is a GC defined over H', $m : H \hookrightarrow G'$ is a monomorphism, $b \in \mathbf{B}$ states whether the graph update u_2 is a canonical graph update, and (u_2, b) is a pair returned by $\mathcal{R}\text{epair}_{\text{add}}(u, f, \phi, m)$,
then $b = \mathit{true}$ iff $u_2 \in \mathcal{S}^{\text{upd}}_{\Delta\text{pres}}(u)$.

In our running example, the local graph repair operation $\mathcal{R}\text{epair}_{\text{add}}$ determines a graph update resulting in the graph marked 2 in Fig. 13. See Fig. 11a for how this local graph repair is obtained using the ST marked by (R2) in Fig. 10d, where the morphism m matches the node a from ψ to the node a_2 in $\mathbf{G}'_{\mathbf{u}}$, but where no extension of m can also match a node of type :B and an edge between these two nodes. The obtained graph update then uses $a \xrightarrow{e} b$ for the graph $\bar{H}$, resulting in the addition of the node b_2 and the edge e_2 from a_2 to b_2.

For $\mathcal{R}\text{epair}_{\text{del}}$, a GC $\exists(f : H \hookrightarrow H', \phi')$ occurring as a subcondition in the consistency constraint ϕ may be satisfied even though it should not be when occurring underneath some negation. Such a violation is determined, again for a given match $m : H \hookrightarrow G'$, by some match $m' : H' \hookrightarrow G'$ satisfying $m = m' \circ f$ and $m' \models_{\text{GC}} \phi'$. The local graph repair operation $\mathcal{R}\text{epair}_{\text{del}}$ (see Fig. 11b for an example) resolves this violation by (a) selecting a graph $\bar{H}$ such that $H \subseteq \bar{H} \subset H'$ using a restriction tree (see Definition 21) where $k' : \bar{H} \hookrightarrow H'$ describes this removal (without loss of generality, we assume that every $f : H \hookrightarrow H'$ used in the consistency constraint is no isomorphism to ensure that suitable graphs $\bar{H}$ exist), (b) extending $m'(H')$ to X_2 such that X_2 contains also all edges (including their source and target nodes) that are connected to nodes in $m'(H') - m'(k'(\bar{H}))$ resulting in $m_2 : X_2 \hookrightarrow G'$, (c) restricting X_2 to X_1 resulting in k'' according to k', (d) computing the pushout complement of k'' and m_2 to remove elements according to k'' from G' to obtain $\ell_2 : G'' \hookrightarrow G'$, and (e) checking whether ℓ_2 removes elements that were added in the provided graph update $u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ using the monomorphism r_1 by checking whether the pullback of r_1 and ℓ_2 is also a pushout. Note that we can't construct the pushout complement of k' and m' as it does not exists when edges are attached to nodes in $m'(H')$ that are not in $m'(H')$.

Definition 35 (Deletion-based Local Graph Repair)
If (see Fig. 16 for a visualization)

- $u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ is a graph update,
- $f : H \hookrightarrow H'$ is a monomorphism,
- $m' : H' \hookrightarrow G'$ is a monomorphism,
- $k' : \bar{H} \hookrightarrow H' \in \text{RT}(H', H)$ is a monomorphism from the restriction tree for H' and H describing a deletion,
- $m_1 : H' \hookrightarrow X_2$ is a monomorphism,
- $m_2 : X_2 \hookrightarrow G'$ is a monomorphism,
- $m' = m_2 \circ m_1$ is a commuting triangle stating that m' is decomposed into m_1 and m_2,
- $k'' \circ m_3 = m_1 \circ k'$ is a commuting square further characterized subsequently,
- X_2 contains the subgraph $m'(H')$ and all edges (including their source and target nodes) that are connected to nodes in $m'(H') - m'(k'(\bar{H}))$,
- X_1 contains the subgraph $k'(m_1(\bar{H}))$ and all nodes of X_2 that are not in $m_1(H')$,
- (m_2, ℓ_2) is the pushout of $(k'', m_4 : X_1 \hookrightarrow G'')$,
- $b \in \mathbf{B}$ is a Boolean recording whether the returned graph repair is a delta preserving graph repair, and
- $b = \mathit{true}$ iff (1) is a pushout and pullback,

then $((\ell_2, \text{id}(G'')), b) \in \mathcal{R}\text{epair}_{\text{del}}(u, f, m')$.

Note that the Boolean value b is used to check whether the graph update obtained by $\mathcal{R}\text{epair}_{\text{del}}$ is delta preserving w.r.t.

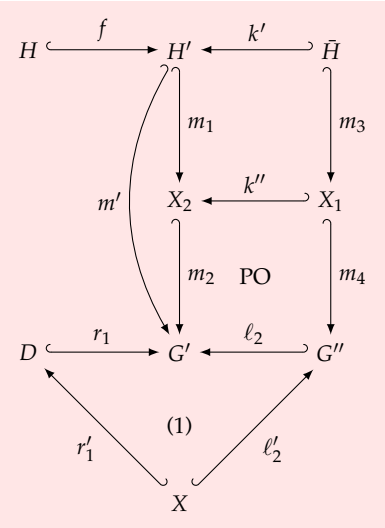


Fig. 16 Visualization for Definition 35

the provided *canonical* graph update u. For our applications, we can safely assume in the following lemma that the given graph update u is canonical because a noncanonical graph update can be converted into an equivalent canonical graph update according to Theorem 2.

Lemma 4 (Deletion-based Local Graph Repair Results in Delta Preserving Graph Updates)
If $u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}_{\text{can}}$ *is a canonical graph update,* u_2 *is a graph update,* $f : H \hookrightarrow H'$ *is a monomorphism,* $m' : H' \hookrightarrow G'$ *is a monomorphism,* $b \in \mathbf{B}$ *states whether the graph update* u_2 *is a canonical graph update, and* (u_2, b) *is in* $\mathcal{R}\text{epair}_{\text{del}}(u, f, m')$,
then $b = true$ *iff* $u_2 \in \mathcal{S}^{\text{upd}}_{\Delta\text{pres}}(u)$.

In our running example, $\mathcal{R}\text{epair}_{\text{del}}$ determines a repair resulting in the graph marked 1 in Fig. 13. See Fig. 11b for a diagram that is used to compute this local repair where we considered the sub-ST marked by (R1) in Fig. 10d where the mono m matches the node a from $\boldsymbol{\psi}$ to the node a_2 in $\mathbf{G}'_{\mathbf{u}}$. The local repair performed then uses $\bar{H} = \emptyset$ for the removal of the node a_2 along with its adjacent loop.

We now consider $\mathcal{R}\text{epair}_{\text{db1}}$, which first computes violations according to Definition 27 and then uses $\mathcal{R}\text{epair}_{\text{add}}$ and $\mathcal{R}\text{epair}_{\text{del}}$ on each of these violations to obtain a local graph repair.

Definition 36 (Single-step Delta-based Graph Repair Algorithm)
If $u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ is a graph update, $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, $\gamma = \text{cst}(\phi, G') \in \mathcal{S}^{\text{GCST}}_{TG,\text{i}(G')}$ is the ST constructed for ϕ and G', and $V = \text{violations}(\gamma, true)$ is the set of violations of γ, then $\mathcal{R}\text{epair}_{\text{db1}}(u, \gamma) = S_1 \cup S_2$ returns graph repairs for addition and deletion using

- $S_1 = \bigcup\{\mathcal{R}\text{epair}_{\text{add}}(u, f, \phi, m) \mid (\oplus, f : H \hookrightarrow H', \phi, m : H \hookrightarrow G) \in V\}$
  and
- $S_2 = \bigcup\{\mathcal{R}\text{epair}_{\text{del}}(u, f, m) \mid (\ominus, f : H \hookrightarrow H', \phi, m : H' \hookrightarrow G) \in V\}$.

For our running example, see again Fig. 13 for the three graph updates obtained in the first step for the ST $\boldsymbol{\gamma}'_{\mathbf{u}}$ and the graph $\mathbf{G}'_{\mathbf{u}}$ from Fig. 10d.

$\mathcal{R}\text{epair}_{\text{db1}}$ indeed generates least changing local graph repairs as stated in the following lemma.

Lemma 5 ($\mathcal{R}\text{epair}_{\text{db1}}$ Generates the Least Changing Local Graph Repairs)
If $u = (\ell_1 : D \hookrightarrow G, r_1 : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ *is a graph update,* $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ *is a GC defined over the empty graph,* $\gamma = \text{cst}(\phi, G') \in \mathcal{S}^{\text{GCST}}_{TG,\text{i}(G')}$ *is the ST constructed for* ϕ *and* G'*, and* $V = \text{violations}(\gamma, true)$ *is the set of violations of* γ*, then* $\{u' \mid (u', b') \in \mathcal{R}\text{epair}_{\text{db1}}(u, \gamma)\} = \mathcal{S}^{\text{repair}}_{\text{lcl}}(G, \phi)$.

We now define locally least changing graph repairs by firstly computing the composition of a sequence of least changing local graph repairs where precisely the last graph update in this sequence is a graph repair that results in a consistent graph w.r.t. the consistency constraint at hand and, secondly, retaining only those obtained graph repairs that are least changing graph repairs w.r.t. the obtained set.

For our running example, see Fig. 13 for the two locally least changing graph repairs that are given by the two graphs marked 1 and 4.

Definition 37 (Locally Least Changing Graph Repair)
If

- $G_1 \in \mathcal{S}^{\text{graphs}}_{\text{fin},TG}$ is a finite typed graph,
- $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph,
- S is the set of all spans $(\bar{\ell}, \bar{r})$ s.t.
  - there is a non-empty finite sequence $\pi = (\ell_1 : D_1 \hookrightarrow G_1, r_1 : D_1 \hookrightarrow G_2) \dots (\ell_n : D_n \hookrightarrow G_n, r_n : D_n \hookrightarrow G_{n+1})$ s.t.
  - $(\ell_i : D_i \hookrightarrow G_i, r_i : D_i \hookrightarrow G_{i+1}) \in \mathcal{S}^{\text{repair}}_{\text{lcl}}(G_i, \phi)$ (for each $1 \leq i \leq n$),
  - $G_i \not\models_{\text{GC}} \phi$ (for each $1 \leq i \leq n$),
  - $G_{n+1} \models_{\text{GC}} \phi$, and
  - $(\bar{\ell} : D \hookrightarrow G_1, \bar{r} : D \hookrightarrow G_{n+1})$ is the iterated composition of the graph updates in π, and
- $(\ell, r) \in \mathcal{S}^{\text{repair}}_{\text{lc}}(G, \phi, S)$,

then (ℓ, r) *is a locally least changing graph repair of* G_1 *w.r.t.* ϕ, written $(\ell, r) \in \mathcal{S}^{\text{repair}}_{\text{llc}}(G_1, \phi)$.

We now define the recursive algorithm $\mathcal{R}\text{epair}_{\text{db}}$ generating such locally least changing graph repairs. It takes the most recent graph update u from graph G to graph G', the ST for

the graph G before the graph update u was applied, and a Boolean instrumentation parameter b_Δ that specifies when it is *true* that only graph repairs should be computed that are also delta-preserving graph updates. The returned set of tuples (u', γ', b') contains graph repairs u' to be applied on G' resulting in graphs G'' where γ' is the ST for the resulting graph G'' and where b' indicates when it is *true* that the graph repair is a delta-preserving graph update.

Technically, the recursive delta-based repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ (a) uses the operation ppgU to propagate the given ST γ across the provided graph update to obtain the ST $\bar{\gamma}$, (b) checks whether the obtained ST $\bar{\gamma}$ is satisfied and returns in this case only the identity graph repair in this case to ensure stability (see our discussion on this at the end of Sect. 5), (c) for the case of non-satisfaction of $\bar{\gamma}$, uses $\mathcal{R}\text{epair}_{\text{db1}}$ to compute all local graph repairs resulting in set S_1 and filters those that are delta-preserving graph updates w.r.t. the provided graph update u when $b_\Delta = \mathit{true}$ (note that graph updates that are not delta-preserving could be repaired later on but that the same graph repair could then be obtained using a sequence with fewer local graph repairs), (d) propagates the ST $\bar{\gamma}$ across each local graph repair in S_1 to obtain the set S_2 of pairs (u', γ', b') with local graph repair u', propagated ST, and preserved Boolean b', (e) constructs the set S_3 of tuples of local graph repairs u', resulting STs γ', and preserved Booleans b' for the case of when the local graph repair u' was successful in establishing consistency, and (f) constructs the set S_4 from the pairs $(u', \gamma', b') \in S_4$ that do not represent successful local graph repairs by (f1) applying $\mathcal{R}\text{epair}_{\text{db}}$ recursively on these elements (composing the graph update u to ensure that its modifications are also preserved when delta-preservation is required) and (f2) composing the successful repairs obtained using this recursive call with the local graph repair u' at hand.

Definition 38 (Delta-based Graph Repair Algorithm)
If G and G' are finite typed graphs from $\mathcal{S}^{\text{graphs}}_{\text{fin},TG}$, $u = (\ell : D \hookrightarrow G, r : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ is a graph update, $b_\Delta \in \mathbf{B}$ is a Boolean that determines whether only delta-preserving graph repairs are to be constructed, $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, $\gamma = \text{cst}(\phi, G) \in \mathcal{S}^{\text{GCST}}_{TG,\text{i}(G)}$ is the ST constructed for ϕ and G, and $\bar{\gamma} = \text{ppgU}(\gamma, u)$ is the propagation of γ over u, then $\mathcal{R}\text{epair}_{\text{db}}(u, \gamma, b_\Delta) = S$, if one of the following items applies.

- $\models_{\text{ST}} \bar{\gamma}$ *(case of satisfaction)*
  and $S = \{((\text{id}(G'), \text{id}(G')), \bar{\gamma}, \mathit{true})\}$.
- $\not\models_{\text{ST}} \bar{\gamma}$ *(case of non-satisfaction)*,
  $S_1 = \{(u', b') \in \mathcal{R}\text{epair}_{\text{db1}}(u, \bar{\gamma}) \mid b_\Delta \rightarrowtail b'\}$ *(all single step local graph repairs)*,
  $S_2 = \{(u', \text{ppgU}(\bar{\gamma}, u'), b') \mid (u', b') \in S_1\}$ *(all single step local graph repairs with propagated ST)*,
  $S_3 = \{(u', \gamma', b') \mid (u', \gamma', b') \in S_2 \wedge \models_{\text{ST}} \gamma'\}$ *(the local graph repairs with propagated ST from S_2)*,
  $S_4 = \{(u'' \circ u', \gamma'', b' \wedge b'') \mid$
  $(u', \gamma', b') \in S_2 \wedge \not\models_{\text{ST}} \gamma'$
  $\wedge (u'', \gamma'', b'') \in \mathcal{R}\text{epair}_{\text{db}}(u' \circ u, \gamma', b_\Delta)$
  $\wedge u'' \circ u' \neq \bot\}$ *(add further local graph repairs recursively)*,
  and $S = S_3 \cup S_4$ *(return additional local graph repairs computed)*.

As pointed out before, this computation does not terminate when local graph repairs trigger each other ad infinitum (see again Fig. 6c for an example of such a GC). However, the breadth-first-computation implemented in $\mathcal{R}\text{epair}_{\text{db}}$ gradually computes a set of graph repairs. Note that the problem of detecting GCs that trigger such nonterminating computations is undecidable and, hence, no sound and complete algorithm for detecting such GCs can be obtained.

We finally state that our delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ returns precisely the desired locally least changing graph repairs upon termination. Moreover, it precisely computes the delta-preserving graph updates that are locally least changing graph repairs.

Theorem 8 *(**Functional Semantics of $\mathcal{R}\text{epair}_{\text{db}}$**)*
The graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ is sound *and* complete *w.r.t. (delta preserving) locally least changing graph repairs, upon termination.*
Formally, if $u = (\ell : D \hookrightarrow G, r : D \hookrightarrow G') \in \mathcal{S}^{\text{upd}}$ is a graph update, $\phi \in \mathcal{S}^{\text{GC}}_{TG,\emptyset}$ is a GC defined over the empty graph, and $\gamma = \text{cst}(\phi, G) \in \mathcal{S}^{\text{GCST}}_{TG,\text{i}(G)}$ is the ST constructed for ϕ and G, then

- $\{u' \mid (u', b') \in \mathcal{R}\text{epair}_{\text{db}}(u, \gamma, \mathit{false})\}$
  $= \mathcal{S}^{\text{repair}}_{\text{llc}}(G, \phi)$ *and, moreover,*
- $\{u' \mid (u', b') \in \mathcal{R}\text{epair}_{\text{db}}(u, \gamma, \mathit{true})\}$
  $= \mathcal{S}^{\text{repair}}_{\text{llc}}(G, \phi) \cap \mathcal{S}^{\text{upd}}_{\Delta\text{pres}}(u)$.

9 Evaluation

We now compare the delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ from Sect. 8 and the two state-based graph repair algorithms $\mathcal{R}\text{epair}_{\text{sb},1}$ and $\mathcal{R}\text{epair}_{\text{sb},2}$ as presented in Sect. 6 w.r.t. their resource requirements. As a first step, we discuss their runtime complexity in subsect. 9.1. As a second step, in subsect. 9.2, we evaluate the efficiency of our prototypical implementation of the algorithms $\mathcal{R}\text{epair}_{\text{db}}$ and $\mathcal{R}\text{epair}_{\text{sb},1}$ in the tool AutoGraph demonstrating that delta-based graph repair is much faster compared to state-based

graph repair.[4] Moreover, we also discuss details of our prototypical implementation impacting execution time, which could be resolved in the future. Lastly, in subsect. 9.3, we compare the three presented graph repair algorithms interpreting their characteristics regarding resource requirements (runtime and memory consumption) and functionality.

9.1 Runtime complexity

The three presented graph-repair algorithms do not terminate for all their inputs resulting in an infinite worst-case execution time (see again the GC in Fig. 6c for which the three algorithms do not terminate when trying to repair the empty graph). To discuss the computational complexity of the algorithms in a meaningful way nonetheless, we implicitly consider in this subsection their restriction to inputs for which they terminate. With this restriction in place, we note that the algorithms may generate an exponential number of graph repairs as demonstrated by the example in Fig. 17 triggered by an atomic modification adding a single loop. We now further discuss why each of the three algorithms has an exponential worst-case execution time.

The runtime complexity of the graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ basically depends on its application of the algorithm $\mathcal{A}$ from [39,40]. This algorithm $\mathcal{A}$, which constructs all minimal graphs satisfying a given GC, has an exponential runtime as it internally computes certain overlappings of the graph under repair with graphs that are contained in the consistency constraint. For example, two graphs G_1 and G_2 containing nodes a_1 and a_2 of type A may be overlapped in a_1 and a_2 resulting in a graph containing G_1 and G_2 but only a single node representing a_1 and a_2. Similarly, a pushout of the two initial monomorphisms $\text{i}(G_1)$ and $\text{i}(G_2)$ describes an overlapping of G_1 and G_2 where no graph elements are identified. As another example, the pushout of the two monomorphisms $m_1 : G_0 \hookrightarrow G_1$ and $m_2 : G_0 \hookrightarrow G_2$ results in an overlapping where G_1 and G_2 overlap in the elements of G_0. The number of such overlappings is exponential in the size of G_1 and G_2 and, moreover, these overlappings are computed in algorithm $\mathcal{A}$ recursively resulting in a tree of computed overlappings (which grows exponentially in width in the example given in Fig. 17). While the algorithm $\mathcal{A}$ is thereby only suitable for GCs with reasonably small graphs, it gradually generates the mentioned sound and complete set of minimal graphs satisfying a given GC. Hence, $\mathcal{R}\text{epair}_{\text{sb},1}$ can only be applied to small graphs given a limited timeframe.

The runtime complexity of the graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ is even worse compared to the runtime complexity of $\mathcal{R}\text{epair}_{\text{sb},1}$ because $\mathcal{R}\text{epair}_{\text{sb},1}$ is applied to each subgraph of the graph under repair in the worst case. This means that (a) the restriction tree (see Definition 21) of the graph under repair *may* need to be constructed entirely (resulting in a restriction tree that is exponential in the size of the graph under repair) and (b) $\mathcal{R}\text{epair}_{\text{sb},1}$ is applied to each subgraph obtained using this procedure. See Fig. 9 for a restriction tree where we need to apply $\mathcal{R}\text{epair}_{\text{sb},1}$ for 11 out of 15 subgraphs. Again, $\mathcal{R}\text{epair}_{\text{sb},2}$ can only be applied to even smaller graphs compared to $\mathcal{R}\text{epair}_{\text{sb},1}$ when runtime is a critical factor.

The runtime complexity of the graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ depends on (a) the offline construction of the map M mapping each subcondition of the consistency constraint to a set of minimal graphs, (b) the offline construction of the ST for the initial graph before graph updates are applied, (c) the propagation of the ST for a given graph update, (d) the construction of graph repairs for the graph update and the propagated ST, and (e) the propagation of the ST over a selected graph repair.

For step (a), we need to apply the algorithm $\mathcal{A}$ from [39,40] but we expect consistency constraints to be written by humans resulting in reasonably small consistency constraints, with an acceptable number of subconditions using sufficiently small graphs. In principle, this step has already an exponential runtime but with the assumption on small consistency constraints and an offline execution, we expect that this step is not likely to be the dominating factor when employing the delta-based graph repair approach based on $\mathcal{R}\text{epair}_{\text{db}}$.

For step (b), we need to recursively compute all matches that may play a role in a satisfaction proof when considering the given initial graph and the consistency constraint. As pointed out at the end of Sect. 3, the computation of these matches requires an exponential amount of time but with standard heuristics for typed graphs, the runtime is often acceptable. Again, since this initial ST construction is performed offline, we expect that the runtime required for this step is tolerable whenever the following steps are sufficiently fast.[5]

For step (c), we need to propagate the ST backwards and then forwards for the two monomorphisms describing the removal and the addition of graph elements given by the graph update. The backward propagation deletes matches from the ST that match graph elements removed by the graph update. Note that we can determine whether a considered

[4] Note that $\mathcal{R}\text{epair}_{\text{sb},2}$ is slower than $\mathcal{R}\text{epair}_{\text{sb},1}$ as discussed in subsect. 9.1 and we therefore only compare $\mathcal{R}\text{epair}_{\text{db}}$ with $\mathcal{R}\text{epair}_{\text{sb},1}$.

[5] In our prototypical implementation, we provide special support for the case when the graph under repair and the graphs in the consistency constraint are all connected: in this case, the construction of STs can be performed more efficiently since partial matches of graph patterns from the consistency constraint can be extended to matches using local search throughout the recursive construction of the ST.

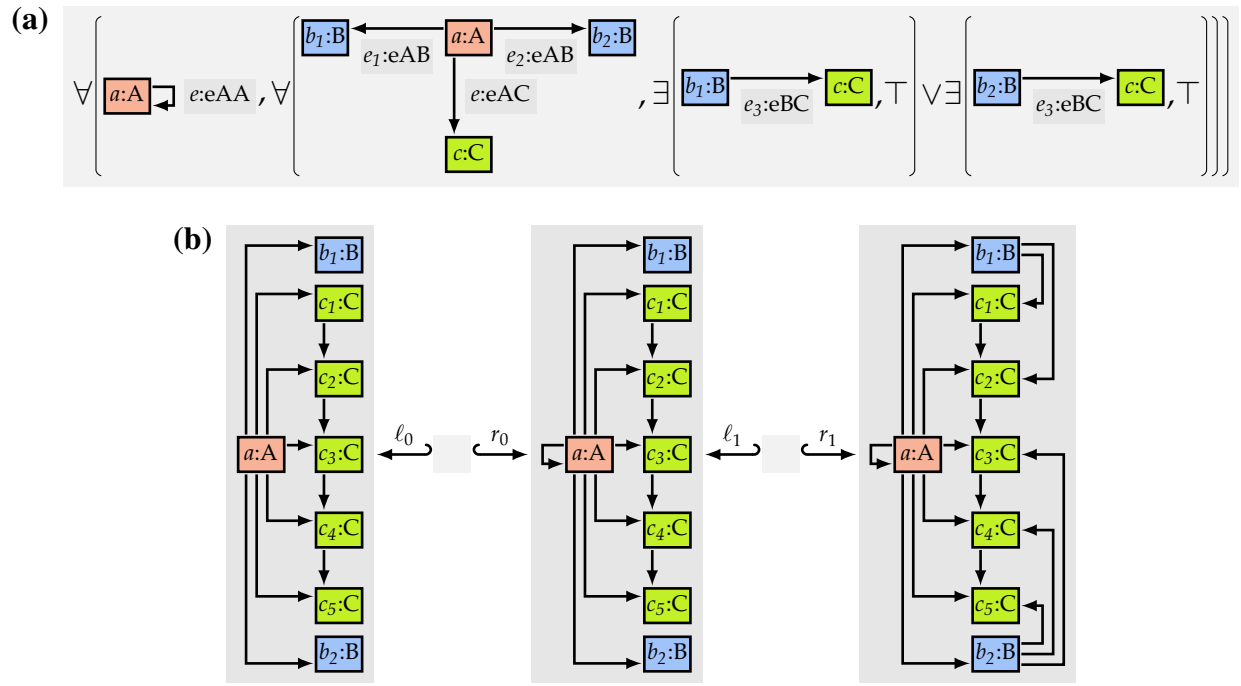


Fig. 17 An example for an exponential number of least changing local graph repairs. **a** A GC representing a consistency constraint stating that if two :B nodes and a :C node are connected to an :A node that has a self-loop, then at least one of the :B nodes is connected to the :C node. **b** A local graph repair (ℓ_1, r_1) for the graph update (ℓ_0, r_0) and the consistency constraint from **a** where edge types have been omitted for readability. In general, when the graph contains n nodes of type C instead of just 5, there are 2^n graph repairs just adding edges between contained nodes. Further local graph repairs may add additional C nodes with suitable edges. Moreover, when delta-preservation is not required, the self-loop can be removed in another graph repair

match must be removed from the ST with linear cost in the number of matches that need to be considered and deleted elements.[6] The forward propagation may result in the computation of *additional* matches that must match at least one added graph element. However, when only a small number of graph elements is added, only a small number of matches need to be constructed.[7]

For step (d), we iteratively compute least-changing and (when required by the user using the corresponding parameter) delta-preserving local graph repairs. The algorithm computes sequences of local graph updates using the operation $\mathcal{R}\text{epair}_{\text{dbl}}$ leading to a local graph repair each.[8] Essentially, we compute a directed acyclic graph (DAG) of such local graph updates as in Fig. 13 where local graph updates leading to graphs earlier obtained may be discarded.[9] In the construction of this DAG, we may obtain a set of local graph updates that is exponential in the size of the graph under repair for each violation that is to be repaired (cf. Fig. 17 again). Also note that each graph update computation using the operation $\mathcal{R}\text{epair}_{\text{dbl}}$ involves the check whether the graph update is delta-preserving w.r.t. the last graph update and the local graph updates computed so far on the current path in the DAG.[10]

For step (e), we refer again to step (c), which performs the same procedure for a graph update instead of a computed graph repair.

[6] In our prototypical implementation, we reduce the number of matches to be considered during backward propagation by storing for each ST the set of graph elements matched by any match contained in it, which allows to abort the recursive backward propagation for unaffected sub-STs.

[7] In our prototypical implementation, local search as in step (b) can be used to more efficiently construct STs required for each additional match.

[8] Consider Fig. 17 for an example where each local graph repair is obtained in five steps for each of the five :C nodes.

[9] In our prototypical implementation, this check for isomorphic graph updates according to Definition 13 requires the computation of additional isomorphisms resulting in a runtime leak that may be fixed by an in-situ implementation as discussed in subsection 9.2.

[10] In our prototypical implementation, determining whether a graph update is delta-preserving according to Definition 18 requires the computation of an isomorphism, which results in the same runtime leak as for the check for isomorphic graph updates.

Table 1 Overview of applications of $\mathcal{R}\text{epair}_{\text{db}}$ or $\mathcal{R}\text{epair}_{\text{sb},1}$

Test	Algorithm	DFS/BFS	Delta-preservation	Suitable fit
Test 1 (noise test)	$\mathcal{R}\text{epair}_{\text{db}}$	DFS	Yes	Quadratic
	$\mathcal{R}\text{epair}_{\text{db}}$	DFS	No	Quadratic
	$\mathcal{R}\text{epair}_{\text{db}}$	BFS	Yes	Quadratic
	$\mathcal{R}\text{epair}_{\text{db}}$	BFS	No	Quadratic
	$\mathcal{R}\text{epair}_{\text{sb},1}$	DFS	—	Unclear*
	$\mathcal{R}\text{epair}_{\text{sb},1}$	BFS	—	Exponential*
Test 2 (violations test)	$\mathcal{R}\text{epair}_{\text{db}}$	DFS	Yes	Cubic
	$\mathcal{R}\text{epair}_{\text{db}}$	DFS	No	Linear
	$\mathcal{R}\text{epair}_{\text{db}}$	BFS	Yes	Cubic
	$\mathcal{R}\text{epair}_{\text{db}}$	BFS	No	Cubic
	$\mathcal{R}\text{epair}_{\text{sb},1}$	DFS	—	Cubic*
	$\mathcal{R}\text{epair}_{\text{sb},1}$	BFS	—	Cubic*

∗: only small number of (comparably large) values

We conclude that efficiency analysis and optimizations for $\mathcal{R}\text{epair}_{\text{db}}$ should therefore focus on the two online steps of ST propagation and computation of local graph repairs.

9.2 Tool-based evaluation

We now report on a tool-based evaluation performed on the basis of our prototypical implementation in the tool AutoGraph. We performed two tests for the state-based algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$, the delta-based algorithm $\mathcal{R}\text{epair}_{\text{db}}$ requiring delta-preservation, and the delta-based algorithm $\mathcal{R}\text{epair}_{\text{db}}$ without requiring delta-preservation. We omit a performance evaluation for the state-based algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ as we are primarily interested in showing that state-based graph repair may be infeasible even for rather small graphs (recall that $\mathcal{R}\text{epair}_{\text{sb},1}$ is less expensive compared to $\mathcal{R}\text{epair}_{\text{sb},2}$).[11]

For each test, we used a machine with 8 GB memory and an i5-4570 CPU @ 3.4 GHz. Subsequently, we discuss the inputs of the two test cases, which are given by a GC specifying a consistency constraint and a most recent graph update that resulted in an inconsistent graph, which depends on a size parameter N. This parameter N affects the most recent graph update in a way that makes graph repair computationally more expensive with growing N.[12] For each element of this sequence, we applied the corresponding graph repair algorithm (i.e., $\mathcal{R}\text{epair}_{\text{db}}$ or $\mathcal{R}\text{epair}_{\text{sb},1}$) 10 times and then computed the average time needed in 30 further runs. See Table 1 for an overview of the applications of both graph repair algorithms where the column *suitable fit* contains information on which kind of polynomial/exponential regression appears to have an appropriate fit in terms of a small *residual sum of squares*. We employed depth-first search (DFS) as well as breadth-first search (BFS) for each case as follows. We employed DFS to allow for a fair comparison of the runtimes of $\mathcal{R}\text{epair}_{\text{db}}$ and $\mathcal{R}\text{epair}_{\text{sb},1}$: since they do not generate the same set of graph repairs in general, we have opted to use DFS returning at most one graph repair. We also employed BFS as described in the previous sections, as BFS guarantees the eventual computation of a graph repair when one exists. However, since BFS also results in a terminating computation for the considered tests, we can conclude that DFS also terminates no matter which computation path is chosen (for the given two examples here). Hence, for the considered tests, DFS comes with a reduced memory footprint and generates a graph repair faster compared to BFS. To evaluate how the use of DFS and BFS affects their runtime, we also applied $\mathcal{R}\text{epair}_{\text{db}}$ and $\mathcal{R}\text{epair}_{\text{sb},1}$ using BFS in both tests. Lastly, for the delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$, we also applied two executions in which we do and do not require delta-preservation. Hence, we obtain the 12 test cases from Table 1.

In the first test (also called *noise-test* subsequently), we checked how a growing host-graph impacts runtime of the graph repair algorithms when only a single violation needs to be repaired. Ideally, graph repair does not depend on the size of the host graph but only on the number of violations that need to be repaired, i.e., the additional elements (related to the parameter N) were just noise in this test, which should not affect the computation. We used the consistency constraint

[11] However, note that $\mathcal{R}\text{epair}_{\text{sb},1}$ and $\mathcal{R}\text{epair}_{\text{sb},2}$ may perform much better than in the following tests depending on the consistency constraints and graphs under repair. Also, the runtime for these two algorithms is dominated by the call to $\mathcal{M}$ and, hence, every optimization of $\mathcal{M}$ would directly translate to the two state-based algorithms. An example of a possible application would be a given initial graph of a graph transformation system that is to be repaired w.r.t. a set of consistency constraints. Such a graph may often be of a rather small size and obtaining all possible graph repairs may be crucial.

[12] In the first test, we add N noise patterns. In the second test, we add N patterns that need repair.

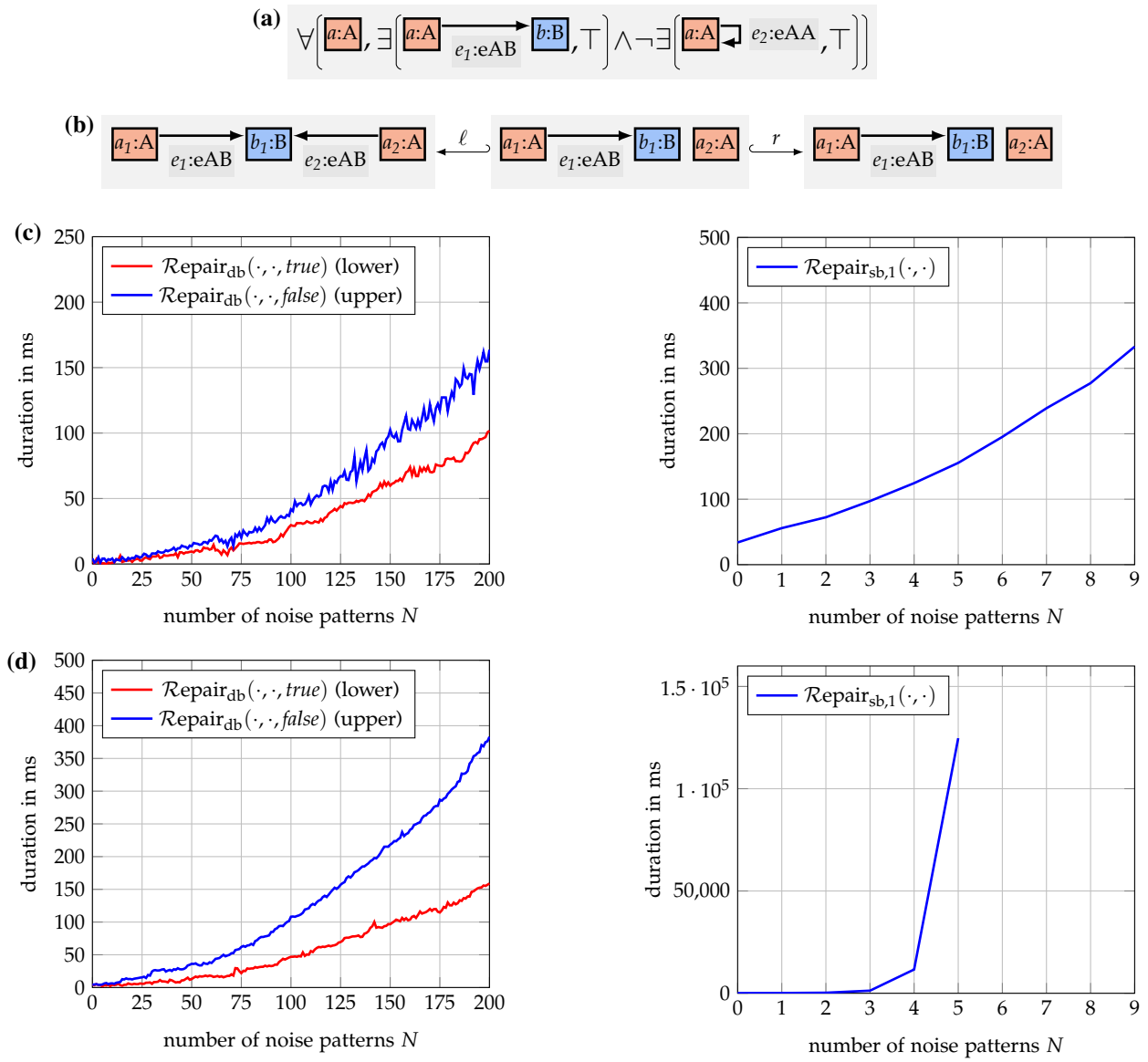


Fig. 18 Applications of $\mathcal{R}\text{epair}_{\text{db}}$ and $\mathcal{R}\text{epair}_{\text{sb},1}$ for testing their resilience against noise. **a** The graph condition used for the test (repeated for readability from Fig. 5b). **b** The graph update used for the test where $N = 0$ graph patterns $a \longrightarrow b$ have been added to all three graphs. **c** Runtime for the delta-based graph repair algorithm with delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, true)$) and without delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, false)$) and the state-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ using depth-first search. **d** Runtime for the delta-based graph repair algorithm with delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, true)$) and without delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, false)$) and the state-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ using breadth-first search

from Fig. 18a (equivalent to the GC ψ from Fig. 5a) and the graph update from Fig. 18b (which is similar to the graph update from Fig. 10a) where N additional copies of the graph pattern $a \longrightarrow b$ are added to each of the three graphs of the graph update. The graph resulting from this graph update can be repaired by (a) adding an edge from a_2 to a new node of type B when delta-preservation is required, (b) adding an edge from a_2 to b_1, or (c) by removing the a_2 node when $\mathcal{R}\text{epair}_{\text{db}}$ is used.

For this first test, the runtimes of the six applications are given in Fig. 18. Firstly, $\mathcal{R}\text{epair}_{\text{sb},1}$ is faster using DFS than when using BFS (which is expected because BFS has to compute all results and because the tableau-based procedure also has to follow computation paths that do not lead to graph repairs) but runtime grows with the number N quite fast in both cases. Secondly, $\mathcal{R}\text{epair}_{\text{db}}$ is much faster than $\mathcal{R}\text{epair}_{\text{sb},1}$ for DFS as well as for BFS, which also holds for both cases of (not) requiring delta-preservation (see different x-axes). For $\mathcal{R}\text{epair}_{\text{db}}$, we observe that requiring

delta-preservation reduces runtime a little bit (by preventing some additional computations that would lead to further graph repairs). Moreover, $\mathcal{R}\text{epair}_{\text{db}}$ is slower using BFS than when using DFS (which is expected since the test using DFS only computes one graph repair whereas the test using BFS computes the complete set of graph repairs). Also, note that $\mathcal{R}\text{epair}_{\text{db}}$ returns three results using BFS and only one result using DFS and, hence, when considering the numbers, the runtime of BFS is about three times higher than the runtime of DFS.

Based on these results, we performed some profiling to determine why $\mathcal{R}\text{epair}_{\text{db}}$ has no constant runtime in this test. Firstly, the ST propagation requires some time growing linearly with N where we need to determine which sub-STs need to be adapted (note that each of the N additional noise patterns results in a match recorded in the ST). Secondly, graph repairs are not computed in-situ in our implementation of $\mathcal{R}\text{epair}_{\text{db}}$, which means that clones of the graphs and STs are constructed throughout the algorithm leading to another linear factor. These two factors result together in an *apparently* quadratic curve in all four applications of $\mathcal{R}\text{epair}_{\text{db}}$.

In the second test (also called *violations-test* subsequently), we checked how multiple violations that must be repaired sequentially impact the runtime of the graph repair algorithms. Ideally, graph repair depends linearly on the number of local violations for delta-based graph repair, i.e., each further local graph repair (related to the parameter N) takes the same amount of time (i.e., N describes in this test how many violations need to be repaired). We used the consistency constraint from Fig. 19a and the graph update from Fig. 19b where N nodes of type B (each connected to a predecessor) are contained in the resulting graph. The graph resulting from this graph update can be repaired by adding a loop edge to each node of type B. Also, when $\mathcal{R}\text{epair}_{\text{db}}$ is used and delta-preservation is not required, some of the given edges may be removed to shorten the length of the chain to be repaired by adding such loops to nodes of type B.

For this second test, the runtimes of the six applications are given in Fig. 19. Firstly, $\mathcal{R}\text{epair}_{\text{sb},1}$ exhibits the same runtime using DFS and BFS probably due to the fact that only one repair (only adding elements) can be obtained.[13] This fact also indicates that $\mathcal{R}\text{epair}_{\text{sb},1}$ performs no relevant amount (if at all) of irrelevant computations in the case of BFS for the considered example. Secondly, as in the first test, the algorithm $\mathcal{R}\text{epair}_{\text{db}}$ is much faster than $\mathcal{R}\text{epair}_{\text{sb},1}$ for DFS as well as BFS, which holds again for both cases of (not) requiring delta-preservation (see different x-axes). For $\mathcal{R}\text{epair}_{\text{db}}$, we also observe that not requiring delta-preservation results (a) for the case of using DFS in an *apparently* constant runtime (since the removal of the edge from a_1 to b_1 is a viable graph repair as well) and (b) for the case of using BFS in an *apparently* cubic runtime when using BFS (stemming from the *apparently* quadratic curve from the previous test with the expected additional linear factor from the number of violations to be repaired). Lastly, $\mathcal{R}\text{epair}_{\text{db}}$ using BFS is just as fast as DFS when requiring delta-preservation for the considered example, which is due to the fact that only one computation path is to be followed to obtain the unique graph repair. As for $\mathcal{R}\text{epair}_{\text{sb},1}$, this indicates that $\mathcal{R}\text{epair}_{\text{db}}$ also executes basically the same computations for DFS and BFS for the considered test.

We believe that our prototypical implementation of $\mathcal{R}\text{epair}_{\text{db}}$ in the tool AutoGraph can be improved in some aspects (resulting in a constant and linear runtime in the first and second test instead of a quadratic and cubic runtime). Firstly, the local graph repairs resulting in a DAG as in Fig. 13 may be computed in parallel where each process generates one path through that DAG independently. Secondly, an in-situ implementation in which graph inclusions are represented throughout the implementation by storing only the added/deleted graph elements should speed up our implementation by fixing the runtime leaks, which lead to the increased runtime of $\mathcal{R}\text{epair}_{\text{db}}$ in the two tests performed. For example, such a reimplementation would not require the cloning of STs and graphs and would improve the runtime of the steps of checking for isomorphic graph updates, checking canonical graph updates, checking for delta-preserving graph updates, and composing graph updates. Thirdly, as a minor change, we believe that the computation of all local graph repairs is often not necessary and we expect that users may specify an upper bound on local graph repairs to be constructed.

9.3 Resource requirements and functionality

For a comparison of the three algorithms, we now summarize our results.

Firstly, for soundness, we have verified for each of the three algorithms that the graph updates computed are indeed graph repairs resulting in consistent graphs.

Secondly, for completeness, we note that each of the three algorithms computes a different set of graph repairs: where the algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ derives a superset of those computed by $\mathcal{R}\text{epair}_{\text{sb},1}$ and where the graph repairs obtained using $\mathcal{R}\text{epair}_{\text{db}}$ are incomparable to the two former sets. This is due to the local nature of the graph repairs computed by $\mathcal{R}\text{epair}_{\text{db}}$ where the locality is induced by the provided consistency constraint. Hence, some graph repairs computed by $\mathcal{R}\text{epair}_{\text{sb},1}$ and $\mathcal{R}\text{epair}_{\text{sb},2}$ are not returned by $\mathcal{R}\text{epair}_{\text{db}}$ when the global perspective on the graph is required. As an example, the graph repair leading to the graph marked 4 in Fig. 9 is

[13] Note that the GC used in this second test (as opposed to the GC in the first test) has been carefully constructed such that $\mathcal{R}\text{epair}_{\text{sb},1}$ computes only a small number of nonessential overlappings for each node of type B resulting here in an *apparently* nonexponential runtime.

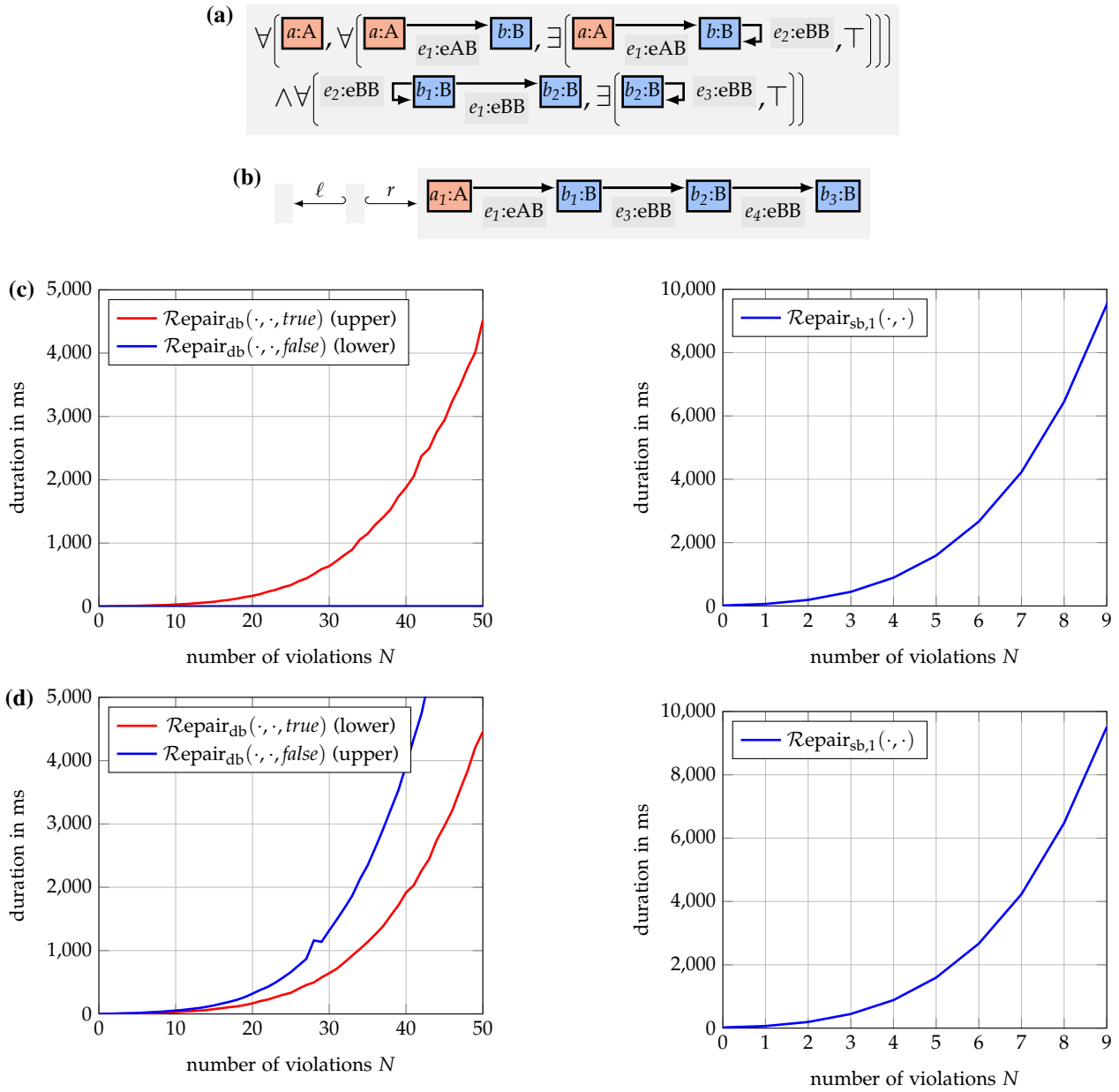


Fig. 19 Applications of $\mathcal{R}\text{epair}_{\text{db}}$ and $\mathcal{R}\text{epair}_{\text{sb},1}$ for testing their resilience against multiple violations. **a** The graph condition used for the test. **b** The graph update used for the test ($N = 3$ nodes of type B are added here). **c** Runtime for the delta-based graph repair algorithm with delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, \mathit{true})$) and without delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, \mathit{false})$) and the state-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ using depth-first search. **d** Runtime for the delta-based graph repair algorithm with delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, \mathit{true})$) and without delta-preservation (i.e., $\mathcal{R}\text{epair}_{\text{db}}(\cdot, \cdot, \mathit{false})$) and the state-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},1}$ using breadth-first search

such a graph repair that is obtained by $\mathcal{R}\text{epair}_{\text{sb},2}$ but not by $\mathcal{R}\text{epair}_{\text{db}}$ as it requires a global view. The degree of locality used for the delta-based graph repair is given by the graph patterns used in the consistency constraint. To increase the number of graph repairs, it is therefore possible to explicitly employ bigger graph patterns in *equivalent* consistency constraints to ensure the additional computation of less-local graph repairs. Hence, the user has the freedom to determine a suitable degree of locality that results in a tradeoff between required runtime and the number/kinds of repairs that are computed. For example, the condition $\boldsymbol{\psi}$ may be rephrased into $\boldsymbol{\psi}' = \boldsymbol{\psi} \wedge \forall(ab, \exists(a \xrightarrow{e} b, \top))$ to also obtain the graph repair marked 4 in Fig. 9. However, in this adapted consistency constraint $\boldsymbol{\psi}'$, the entire graph would need to be

checked for all combinations of an :A node and a :B node, which would result in an increased runtime.

Thirdly, as pointed out in subsect. 6.3 and as supported by our tool-based evaluation in the previous subsection, the two state-based algorithms have an undesirable runtime especially when used for online repair in a scenario where the graph at hand is subject to a sequence of graph updates that invalidate consistency over and over again. The delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ on the other hand has proven to be much faster compared to the state-based graph repair algorithms according to our evaluation. As a side note, we point out that a brute-force algorithm enumerating all graph updates (e.g. sorted by the number of atomic changes) will also be able to generate the same set of graph repairs as generated by $\mathcal{R}\text{epair}_{\text{sb},2}$ but such an algorithm would exhibit a much worse runtime compared to $\mathcal{R}\text{epair}_{\text{sb},2}$, which computes these graph repairs in a systematic way exploring only modifications that may lead to graph repairs subsequently.

However, due to the expressiveness of the underlying graph logic, the state-based graph repair algorithms do not terminate since AutoGraph may not terminate and the delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ may not terminate (even though it uses AutoGraph only in the offline phase) since an infinite sequence of local graph repairs would be computed. Also note that it is not easy to syntactically restrict the underlying graph logic in a least-restrictive way to obtain terminating graph repair algorithms. For example, the three presented algorithms do not terminate when the empty graph is to be repaired w.r.t. the GC from Fig. 6c as a consistency constraint. This example demonstrates that limiting the nesting depth to 2 is insufficient and, obviously, limiting nesting depth to 1 would drastically reduce the expressiveness of the logic. Also note that the computations of all three algorithms do not get stuck in cycles and do not skip results to be returned: they always proceed in a reasonable direction but may not terminate because they are forced on a path to an infinite graph, which they can't generate by incrementally adding a finite number of elements. Note that the state-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{sb},2}$ exhibited out-of-memory errors in our two tests for $N = 10$.

Lastly, we point out that the ST for a graph and a consistency constraint used in the delta-based graph repair algorithm may need to store a large number of matches therefore requiring a large amount of memory. From this perspective, the delta-based graph repair algorithm is also a tradeoff between execution time and memory consumption. Note that memory consumption was no problem in our tool-based evaluation but further evaluations using bigger graphs and an implementation designed for such bigger graphs may provide further insights in implementation challenges. Anyway, the two state-based graph repair algorithms with their online application of the algorithm $\mathcal{A}$ require much more memory even for the rather small examples presented.

We conclude that the three presented graph repair algorithms determine different choices for the tradeoff between required runtime on the one hand and required memory and completeness w.r.t. the set of all least changing (local) graph repairs on the other hand. Moreover, the novel notion of delta-preserving graph repairs is important for the delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ and helps to further restrict the resulting set of graph repairs to support the user in making a choice between the possible repairs.

10 Case study

For an additional application of the delta-based graph repair algorithm $\mathcal{R}\text{epair}_{\text{db}}$ presented in the previous section, we consider the scenario described in the social network benchmark (SNB) from [44], which is maintained and developed by the Linked Data Benchmark Council (LDBC). This benchmark describes an online social network in which users are members of moderated forums where posts can be liked and commented. The general aim of the benchmark is the evaluation and comparison of approaches for the management of graph-based systems such as query languages and their implementations in query engines. Consider the class diagram given in Fig. 3 of this benchmark where we have omitted node and edge attributes that are not covered yet by our approach. For graphs that are typed over this class diagram, we can express meta-model consistency constraints using GCs.

One of the typical meta-model constraints for class diagrams, also listed in [44, pp. 16–17], relates to multiplicity constraints for relations. We consider here, for simplicity, only the following two multiplicity constraints.

- **M1** Each comment is the source of some :replyOf edge to some comment or post.
- **M2** Each post is the target of some :hasCreator edge from some person.

The encoding of such consistency constraints using GCs is straightforward as already demonstrated previously in [40]. In addition, we consider the informal meta-model consistency constraint given in [44, p. 12], which refers to multiple relations.

- **P** Posts in a forum can be created by a non-member person if and only if that person is a moderator.

Consider the formalization of the property **P** in the form of a GC in Fig. 20a, which is violated in the resulting graph database whenever a moderator or a member cancels a subscription to a forum in which it is the creator of a post.

The consistency constraint "comments of a post are contained in the same forum", which is not included in [44], is also not required for the given class diagram because the

containment relation for comments is not covered explicitly in the class diagram. However, the removal of a post from a forum can then be expected to also invoke the removal of all comments directly or indirectly connected to the removed post. Hence, in a social network of realistic size, the removal of a moderator or member (i.e., removing elements of the :hasMember or :hasModerator relation) with all its posts may affect an enormous amount of graph elements (i.e., nodes of types :Post and :Comment) to re-establish consistency, which may not be desirable in general. For the example at hand, consider the graph update Fig. 20b in which a moderator *Pe1* leaves a forum. The resulting graph is inconsistent w.r.t. the property **P** while it still satisfies the multiplicity constraints **M1** and **M2**.

An application of $\mathcal{R}\text{epair}_{\text{db}}$ for this example now results in several local graph repairs as follows.

- *Local Graph Repair 1*
  - *Action:* Recreation of the e_1:hasModerator edge that was removed by the graph update:
  - *Result:* This local graph repair results in a consistent graph.
  - *Evaluation:* Every graph repair including this local graph repair would not be delta preserving.
- *Local Graph Repair 2*
  - *Action:* Creation of a : hasMember edge from *Pe1* to *F*.
  - *Result:* This local graph repair results in a consistent graph.
  - *Evaluation:* The local graph repair is a locally least changing and delta-preserving graph repair to be returned.
- *Local Graph Repair 3*
  - *Action:* Deletion of the matched *F* : Forum node and all connected edges $\{e_3, e_4\}$.
  - *Result:* This local graph repair results in a consistent graph since we did not include all multiplicity constraint from the benchmark (in particular, we can expect that every post should be contained in a forum by means of an edge of type :containerOf).
  - *Evaluation:* The local graph repair is a locally least changing and delta-preserving graph repair to be returned.
- *Local Graph Repair 4*
  - *Action:* Deletion of the matched *Pe1* : Person node and all connected edges $\{e_2, e_8\}$.
  - *Result:* This local graph repair results in a consistent graph but, again, it contains a post for which no forum is the container.
  - *Evaluation:* The local graph repair is a locally least changing and delta-preserving graph repair to be returned.
- *Local Graph Repair 5*
  - *Action:* Deletion of the matched *Po* : Post node and all connected edges $\{e_2, e_3, e_6, e_7\}$.
  - *Result:* This local graph repair results in a graph that is satisfying **P** but not the multiplicity constraint **M1**.
  - *Evaluation:* Further applications of the single-step algorithm $\mathcal{R}\text{epair}_{\text{db1}}$ would remove also the two nodes *Co1* : Comment and *Co2* : Comment as well as the edges $\{e_5, e_8\}$.
- *Local Graph Repair 6*
  - *Action:* Deletion of the edge e_2:hasCreator. This local graph repair is depicted in Fig. 21.
  - *Result:* This local graph repair results in a graph that is satisfying **P** and M1 but not the multiplicity constraint **M2**.
  - *Evaluation:* Further applications of the single-step algorithm $\mathcal{R}\text{epair}_{\text{db1}}$ would, amongst other local graph repairs that are computed, remove also the node *Po* : Post with the attached edges (this local graph repair is depicted in Fig. 22) and then, amongst other local graph repairs that are computed, also remove the nodes *Co1* : Comment and *Co2* : Comment with attached edges (this local graph repair is depicted in Fig. 23).
- *Local Graph Repair 7*
  - *Action:* Deletion of the edge e_3:containerOf.
  - *Result:* This local graph repair results in a graph satisfying **P** and M1 but not the multiplicity constraint **M2**.
  - *Evaluation:* Further applications of the single-step algorithm $\mathcal{R}\text{epair}_{\text{db1}}$ would remove further elements not leading to additional least-changing delta-preserving graph repairs.

Note that some of these graph repairs may be undesirable as for example the removal of the entire forum. Hence, we argue that graph repair requires in this setting user interaction to determine the desired graph repair.

11 Related work

The recent survey on *model repair* [28] and the corresponding exhaustive classification of primary studies selected in the literature review [27] discusses a huge amount and wide variety of existing approaches that renders a detailed comparison with all of them infeasible.

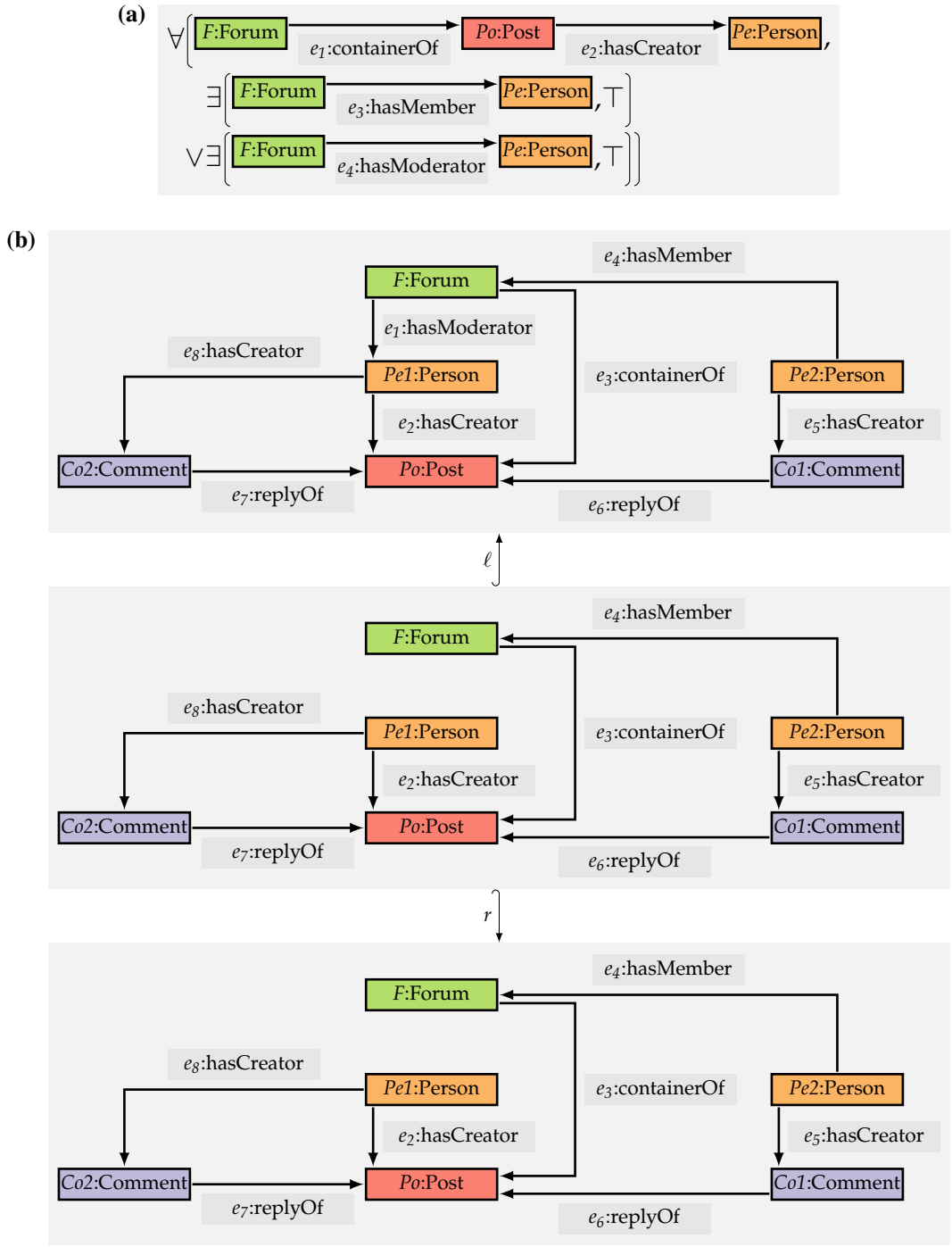


Fig. 20 A consistency constraint **P** formalized as GC and a graph update used in Sect. 10 in an application of the delta-based graph repair algorithm to the social network benchmark (SNB) [44] of the Linked Data Benchmark Council (LDBC). **a** A formalization of the consistency constraint **P** from Sect. 10 stating that "Posts in a forum can be created by a non-member person if and only if that person is a moderator". **b** A graph database update where a moderator *Pe1* leaves a forum in which it has a post *Po*

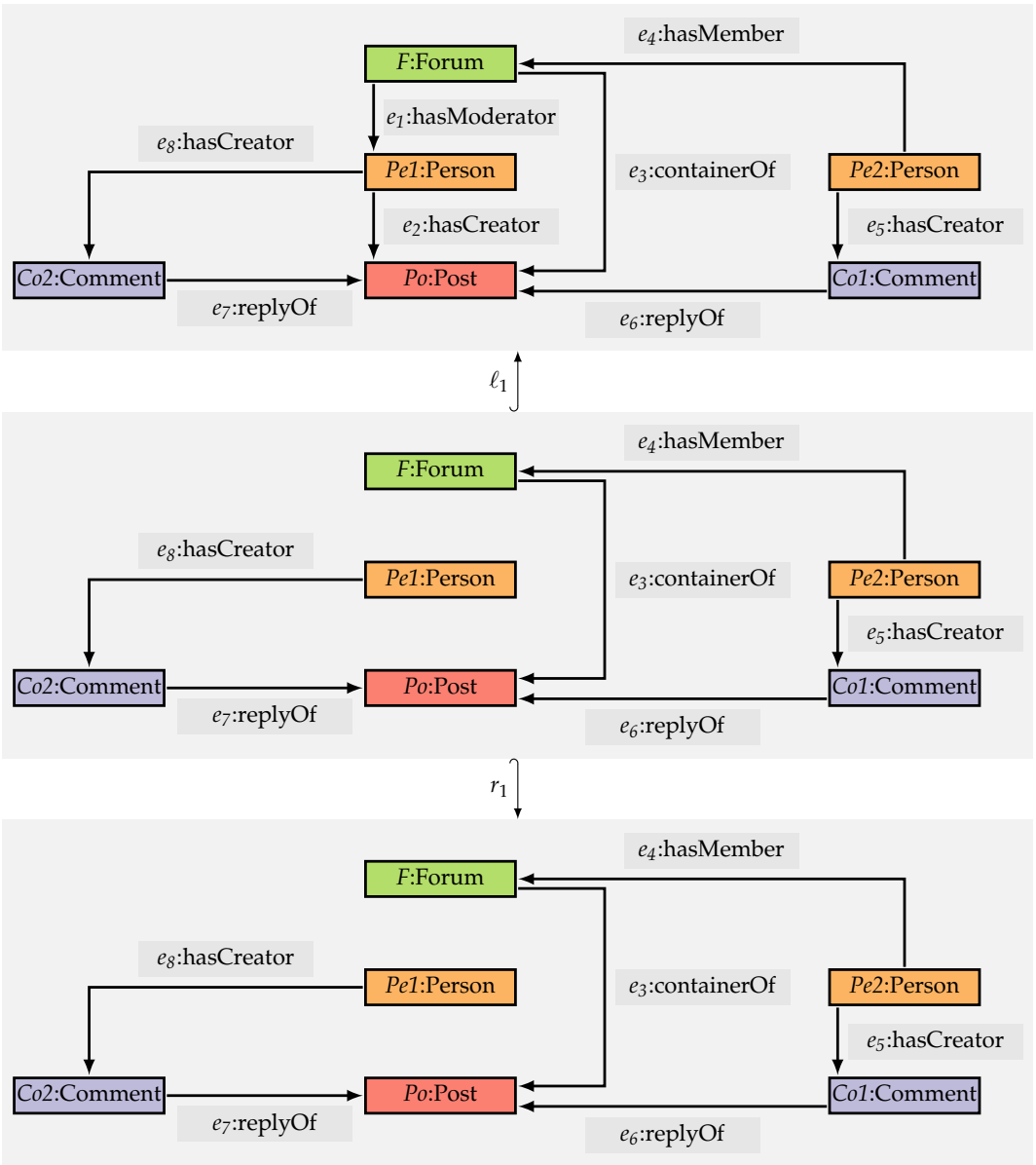


Fig. 21 A local graph repair (ℓ_1, r_1) computed using $\mathcal{R}\text{epair}_{\text{dbl}}$ for the graph update from Fig. 20

We consider our approach to be innovative since it addresses the important issues of *completeness* and *least changing* for incremental graph repair in a precise and formal way.[14] Only two other approaches [26,42] that are mentioned in the survey [27,28] address these two properties and also employ existing constraint-solving technology. However, the main difference with our approach is that these approaches are not incremental. In particular, a logic programming approach is proposed in [42], which allows for the exploration of model repair solutions for re-establishing conformance of a model with its metamodel. These possible repairs are ranked in this approach according to some quality criteria but neither soundness nor completeness of these model repair solutions is formally verified. Moreover, the least changing bidirectional model transformation approach in [26] only obtains repairs of a bounded size by relying on a bounded constraint solver. Also, the least-changing principle in these approaches is based on some distance metrics, counting the number of deletions/additions necessary to reach the other model. Our repair algorithm returns all repairs based on such a minimal distance, but it is more diverse by considering all modifications establishing consistency preserving as many nodes/edges as possible compared to other repairs.

[14] Our approach additionally addresses delta preservation, which is not considered in the survey [27].

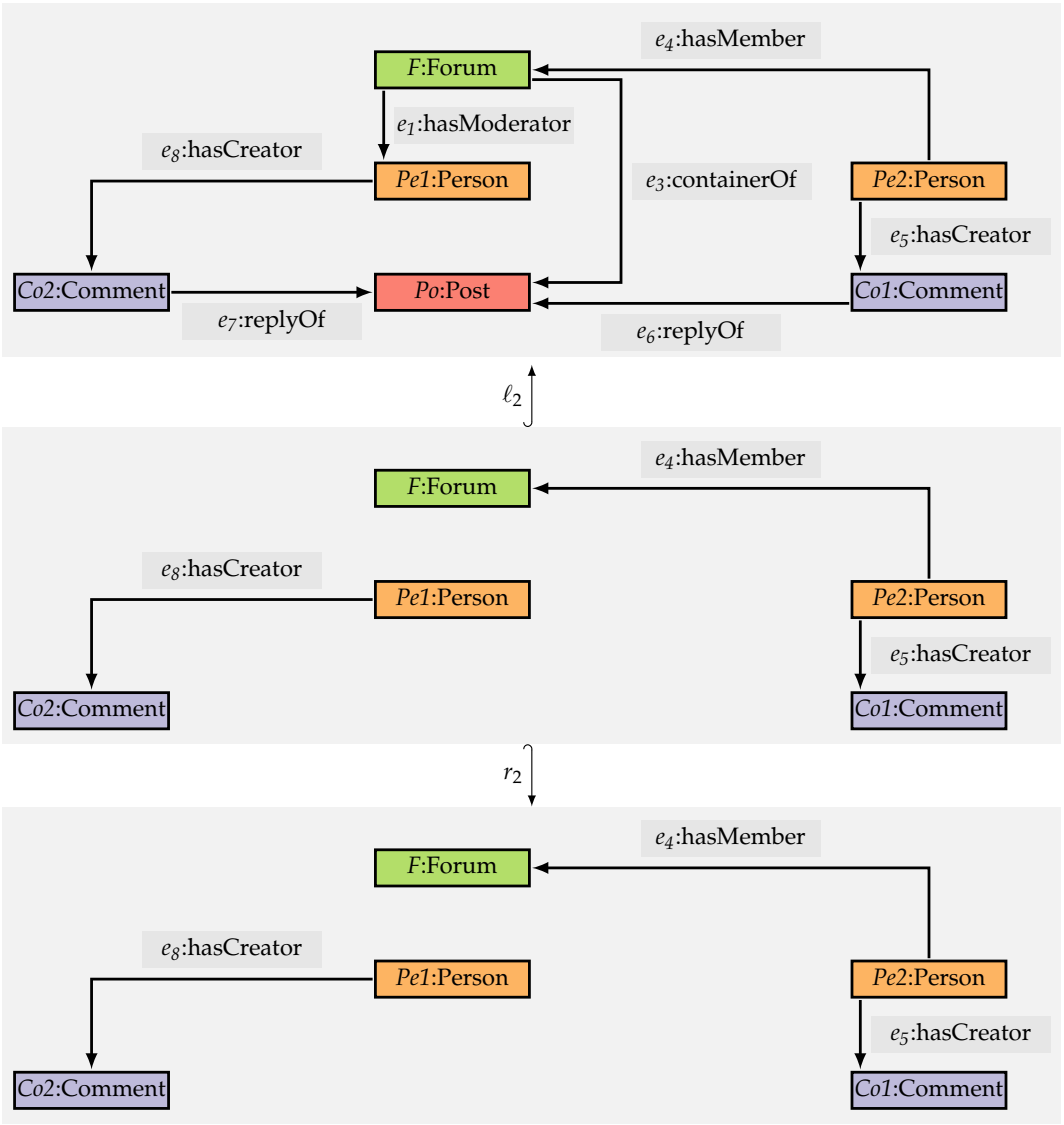


Fig. 22 A local graph repair (ℓ_2, r_2) computed using $\mathcal{R}\text{epair}_{\text{db1}}$ for the graph update from Fig. 20 continuing from the local graph repair from Fig. 21

Some *recent work* on rule-based *graph repair* [20] (not covered by the survey) addresses the least changing principle by developing so-called maximally preserving (items are preserved whenever possible) repair programs. This state-based approach considers a subset of consistency constraints (up to nesting depth 2) handled by our approach, and is not complete, since it produces repairs including only a minimal amount of deletions. The same authors continue this line of research with a modified approach [38], where a repair program is derived from a given set of repair rules. These repairs however in general do not follow the least-changing principle any more. Some other recent rule-based graph repair approach [30,43] (also not covered by the survey) proposes so-called *change preserving repairs* (similar to what we define as delta-preserving). Finally, Cheng et al. [13] also present a rule-based approach to graph repair, where they in particular concentrate on repairing specific properties such as incompleteness, conflicts, and redundancies. The main difference of all these rule-based approaches with our work is that we do not require the repair operations to be given in the form of rules, since we derive repairs using constraint solving techniques directly from the desired consistency constraints. Moreover, most of these rule-based approaches are neither concerned explicitly with the least-changing principle, nor with completeness. Another recent work [47] concentrates on automated repair of Alloy models, which uses a first order relational logic with transitive closure. This approach is in line with traditional automated program repair techniques, using a generate-and-validate technique. A generated model repair is declared to be valid as soon as all tests pass for the

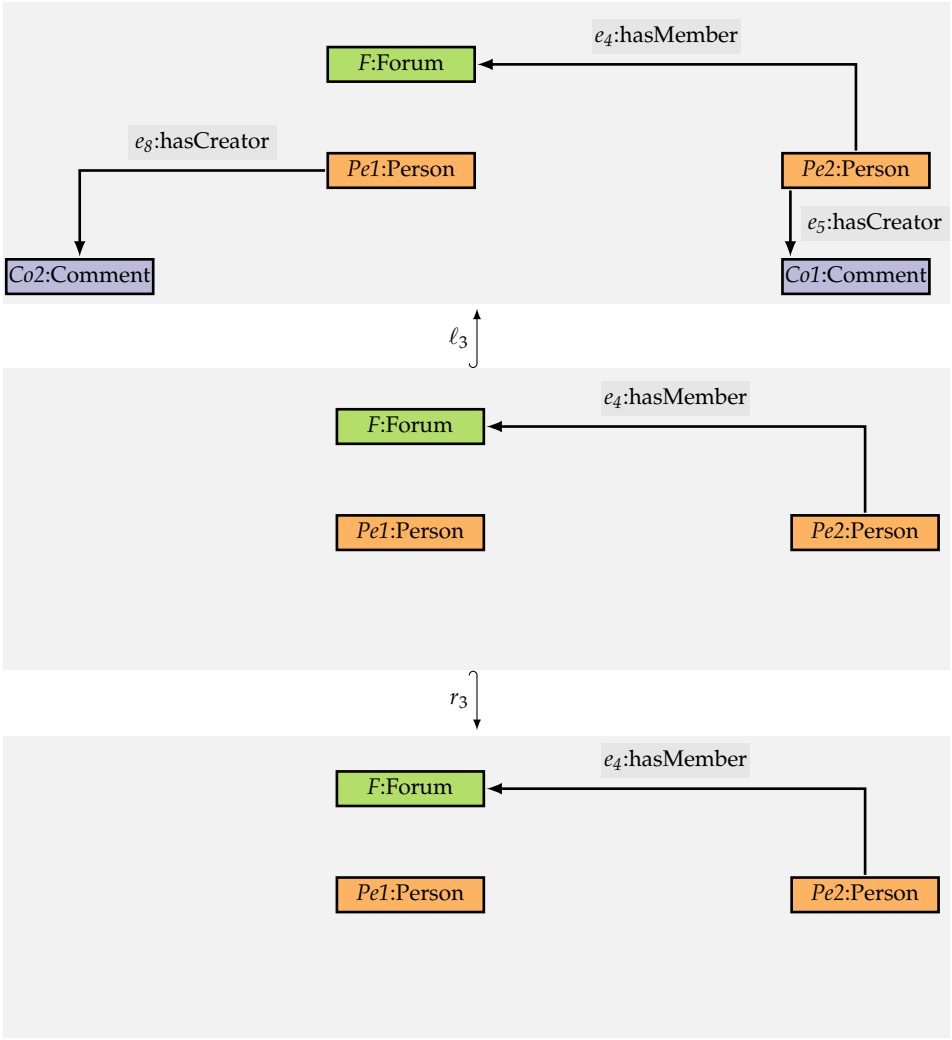


Fig. 23 A local graph repair (ℓ_3, r_3) computed using $\mathcal{R}\text{epair}_{\text{db1}}$ for the graph update from Fig. 20 continuing from the local graph repair from Fig. 22

repaired model. Neither completeness, nor the least-changing principle are addressed explicitly by this approach.

Another line of related work is the research on repair methods for *multi-models*. As described in the model repair survey [27,28] these repair methods provide dedicated support for multi-model scenarios and inter-model consistency constraints (as opposed to the intra-model consistency constraints considered in this paper). A special case of multi-model repair is *model synchronization*. More precisely, in the area of model transformation (see [22]) two or more models are synchronized if they describe (different two views of) the same artifact. In this context, when one or more models are modified such that these models become inconsistent, then model synchronization is the repair process that makes them consistent again. Multi-model techniques often impose restrictions on the user updates as well as on the generated repairs. For example, usually user updates are allowed either on the source or on the target model, whereas the repair is then restricted to the target model or source model, respectively. Moreover, model synchronization and, in general, the inter-model consistency constraints can be described by means of relations (as e.g. in [26] using QVT Relations or as e.g. in [17] using triple graph grammars), or more implicitly by means of unidirectional operational definitions (as e.g. in [10]). Anyhow, the literature on model synchronization and multi-model consistency is also quite large and, as pointed out in [28], even if model repair overlaps with multi-model consistency, there are several topics that are specific for just one of these areas. Our approach could be used in such multi-model scenarios by defining the inter-model consistency constraint by a graph condition that expresses the relation between the different models, but it is up to future work to elaborate a dedicated multi-model procedure and work out the advantages of such a constraint-based approach.

Finally, a wide variety of work on *incremental evaluation of graph queries* (see e.g. [5,7]) aims at supporting an efficient re-evaluation of a given graph query after a graph update has been performed on the graph at hand. Although not employed with the specific aim of complete and least changing graph repair, this work is related to our newly introduced concept

of satisfaction trees, also using specific data structures to record with some detail the set of answers to a given query (as described for graph conditions, for example, also in [6]). It is part of ongoing work to evaluate how STs can be employed similarly in this field of incremental query evaluation.

12 Conclusion

We presented a logic-based incremental approach to graph repair. It is the first approach to graph repair returning a sound and complete overview of the set of least-changing repairs with respect to graph conditions equivalent to first-order logic on graphs. Moreover, our incremental approach has built-in support for delta-preservation, ensuring that only repairs are generated preserving the modifications of the graph update that resulted in the violation of consistency. Technically, our approach relies on an existing model generation procedure for graph conditions together with the newly introduced notion of satisfaction trees, which encode if and how a graph satisfies a graph condition. In particular, the set of violations of a satisfaction tree represents in a detailed way the parts of the consistency constraint that are violated.

As future work, we aim at supporting attributes in consistency constraints. We are confident to be able to realize this extension, since the underlying model generation procedure used for generating repairs supports such graph attributions already. Ongoing work is the support of more expressive consistency constraints, allowing path-related properties. Moreover, we are in the process of evaluating our approach on more case studies. This evaluation also pertains to the overall efficiency (for which we employ techniques for localized and incremental pattern matching) and includes a comparison with other approaches for graph repair. We plan to work out a dedicated multi-model procedure in line with our constraint-based approach by defining the inter-model consistency constraint by a graph condition that expresses the relation between the different models. Finally, we aim at presenting more properties going beyond delta preservation or least-change (e.g. also address least-surprise as elaborated in the field of bidirectional transformations [12]) allowing for an even more diverse distinction between all possible repairs supporting the implementation of a powerful (interactive) repair selection procedure.

Acknowledgements We would like to express our great appreciation for the insightful comments made by the anonymous reviewers, which helped to improve our contribution considerably.

Funding Open Access funding enabled and organized by Projekt DEAL.

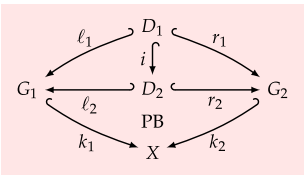
A proofs

We provide proofs for the lemmas and theorems of the main part of the paper.

Proof of Theorem 1: Typed Graphs are a Category See [15, Example 2.12, pp. 25–26]. □

Proof of Theorem 2: Existence of Canonical Graph Update An incremental computation of a strict reduction is guaranteed to terminate for graph updates with a finite graph D resulting in a canonical graph update in these cases. However, since reductions can increase the graph D arbitrarily, we can also deduce that every graph update can be restricted to a canonical graph update in general.

For a direct construction, we obtain the graph D_2 of the canonical graph update $u_2 = (\ell_2, r_2)$ such that it contains the maximally preserved subgraph of the given graph update $u_1 = (\ell_1, r_1)$.



We construct u_2 in two steps.

- We construct a smallest overlapping X of the graphs G_1 and G_2 formally represented by a pair $(k_1 : G_1 \hookrightarrow X, k_2 : G_2 \hookrightarrow X)$ of two jointly epimorphic monomorphisms (i.e., every node or edge of X is mapped to by either k_1 or k_2) satisfying $k_1 \circ \ell_1 = k_2 \circ r_1$.
- We construct the pair (ℓ_2, r_2) as the pullback of this overlapping (k_1, k_2).
- We have $k_1 \circ \ell_2 = k_2 \circ r_2$ from the pullback construction.
- We have a morphism $i : D_1 \rightarrow D_2$ satisfying $\ell_1 = \ell_2 \circ i$ and $r_1 = r_2 \circ i$ by the pullback property.
- Lastly, i is a monomorphism because ℓ_1 and ℓ_2 are monomorphisms. □

***Proof of Theorem 3*: Functional Semantics of $\mathcal{R}\text{epair}_{\text{sb},1}$** The soundness of $\mathcal{R}\text{epair}_{\text{sb},1}$ follows directly from the formal results on AUTOGRAPH from [40]. For completeness consider that $\mathcal{R}\text{epair}_{\text{sb},1}$ only returns repairs (l, r) where l is the

identity. Hence, $\mathcal{R}\text{epair}_{\text{sb},1}$ only returns non-deleting repairs. For the monomorphism r we also rely on the completeness guaranteed by AUTOGRAPH according to from [40]. □

***Proof of Theorem 4*: Functional Semantics of $\mathcal{R}\textbf{epair}_{\textbf{sb,2}}$** Since the definition of $\mathcal{R}\text{epair}_{\text{sb},2}$ ensures that non-least changing repairs are removed prior to returning the derived set, we only have to show that the updates obtained are indeed repairs. This proof proceeds by induction following the traversal of the generates restriction tree $\text{RT}(G, \emptyset)$. The soundness of $\mathcal{R}\text{epair}_{\text{sb},2}$ then again follows immediately from the formal results on AUTOGRAPH from [40] as for $\mathcal{R}\text{epair}_{\text{sb},1}$. Moreover, on the one hand, when we would traverse the entire restriction we would clearly consider all possible restrictions l of the given graph G and, on the other hand, AUTOGRAPH ensures again the completeness of the morphisms r using the repairs. Hence, for completeness we argue that the stopping condition for graphs G_c that satisfy the condition ψ does not limit completeness. For this, observe that every repair that would be obtained by some direct or indirect child of a graph G_c from the restriction tree satisfying ψ would not be least changing due to the repair $(l : G_c \hookrightarrow G, \text{id}(G_c))$ constructed for G_c. □

***Proof of Theorem 5*: Soundness of the (Recursive) Construction of Satisfaction Trees** By induction on ϕ mainly showing that m'_t and m'_f are defined in the case of the *exists* operator for the correct matches $q : H_i \hookrightarrow G$, which follows from the fact that all matches are considered by construction and that the check for satisfaction on the corresponding ST is performed as required.

***Proof of Theorem 6*: Compatibility of Satisfaction of Satisfaction Trees and Computation of Violations** The only step that needs to be verified is that at least one violation is returned iff the ST is not satisfied. We apply induction on the structure of the ST at hand and only consider the case of existential quantification.

If a violation is found, this means that the partial map m_t is not empty and b is *false* or that the partial map m_t is empty and b is *true* according to Definition 27. Correspondingly, a ST is not satisfied when m_t is empty.

This means that a violation is obtained using an application $\text{violations}(\neg\exists(f, \phi, m_t, m_f), \textit{true})$ for elements in m_t, which then results in nonsatisfaction of the ST due to the enclosed negation. Vice versa, if a violation is obtained using an application $\text{violations}(\exists(f, \phi, m_t, m_f), \textit{true})$ this means that m_t is empty, which then also results in nonsatisfaction of the ST. □

***Proof of Lemma 1*: Compatibility of Satisfaction Tree Construction and Backward Propagation** By induction on the common structure of the two STs and ϕ mainly showing that the mappings m_t and m_f are equal in the case of the *exists* operator. This means that they are both defined for the correct matches $q : H_i \hookrightarrow G$, which follows from the fact that no additional matches can be found since l restricts the graph, that only matches are removed that could not be preserved, and that the preserved matches have been inserted in the correct map m'_t or m'_f depending on whether the corresponding ST is satisfied.

***Proof of Lemma 2*: Compatibility of Satisfaction Tree Construction and Forward Propagation** By induction on the common structure of the two STs and ϕ mainly showing that the mappings m_t and m_f are equal in the case of the *exists* operator. This means that they are both defined for the correct matches $q : H_i \hookrightarrow G$, which follows from the fact that all old matches can be preserved and that all additional matches are contained for newly constructed STs, and that the obtained matches have been inserted in the correct map m'_t or m'_f depending on whether the corresponding ST is satisfied. □

***Proof of Theorem 7*: Compatibility of Satisfaction Tree Construction and Update Propagation** From Lemma 1 and Lemma 2. □

***Proof of Lemma 3*: Addition-based Local Graph Repair Results in Delta Preserving Graph Updates** Because no elements are deleted using $\text{id}(G')$, the diagram in Definition 18 is simplified as follows. When X is constructed as the PB of r_1 and ℓ_2, we know that the graph X equals $D_2, r'_1 = \text{id}(D_2)$, $\ell'_2 = r_1$, (2) is also a PO, and ℓ and r can be constructed such that (1) and (3) commute. The only remaining condition to be checked is whether $(\ell_1 \circ r'_1, r_2 \circ \ell'_2) = (\ell_1, r_2 \circ r_1)$ is canonical. By Definition 34 this property is checked and returned as b. □

***Proof of Lemma 4*: Deletion-based Local Graph Repair Results in Delta Preserving Graph Updates** According to Definition 18, we have to construct the pullback of r_1 and ℓ_2 and check whether it is a pushout. This condition is also checked according to Definition 35. Moreover, the composition of both graph updates is canonical as required when the graph update u is canonical, which holds by assumption because the constructed graph update does not add elements by using the monomorphism $\text{id}(G'')$.

***Proof of Lemma 5*: $\mathcal{R}\textbf{epair}_{\textbf{db1}}$ Generates the Least Changing Local Graph Repairs** $\mathcal{R}\textbf{epair}_{\textbf{db1}}$ performs a recursive descent throughout the provided ST to determine sub-STs that are incorrectly violated or incorrectly satisfied. For the case of conjunction, $\mathcal{R}\text{epair}_{\text{db1}}$ considers all possible repairs

of sub-STs independently from each other by selecting one sub-ST that needs repair or by selecting some sub-ST that can be broken to achieve the desired repair result. Repairs are obtained from existential quantifications only: we have the two cases given by $\mathcal{R}\text{epair}_{\text{add}}$ and $\mathcal{R}\text{epair}_{\text{del}}$ discussed below but also the possible recursive cases where violations are resolved for the sub-ST given for some existing matches. Thereby the recursive procedure descents to an arbitrary violation of the given ST leading to completeness when $\mathcal{R}\text{epair}_{\text{add}}$ and $\mathcal{R}\text{epair}_{\text{del}}$ are complete in this respect. These two cases correspond to the two cases for least changing local graph repairs as well as for $\mathcal{R}\text{epair}_{\text{add}}$ and $\mathcal{R}\text{epair}_{\text{del}}$. Subsequently we show that the least changing local graph repairs by addition are obtained using $\mathcal{R}\text{epair}_{\text{add}}$ and that the least changing local graph repairs by deletion are obtained using $\mathcal{R}\text{epair}_{\text{del}}$.

For $\mathcal{R}\text{epair}_{\text{add}}$, we clearly only obtain least changing local repairs by addition where the monomorphism e_2 in the definition of least changing locally graph repairs corresponds to the monomorphism $\bar{m} \circ k$ where in both cases the corresponding pushouts are constructed. Also, due to the construction using AutoGraph and again relying on the formal results from [40], we also obtain completeness with respect to the addition of elements computed in this step using $\mathcal{M}$ beforehand.

For $\mathcal{R}\text{epair}_{\text{del}}$, we clearly only obtain least changing locally repairs by deletion where the monomorphism e_1 in the definition of least changing locally graph repairs corresponds then to the monomorphism m_2. Also, due to the construction of the restriction tree employed in this step, we obtain a complete consideration of possible restrictions of the graph H'. □

***Proof of Theorem 8*: Functional Semantics of $\mathcal{R}\text{epair}_{\text{db}}$** By induction on the recursive execution of $\mathcal{R}\text{epair}_{\text{db}}$, we conclude that only iterated compositions of updates are returned. Moreover, as shown subsequently, these graph updates are locally least changing graph updates as defined in Definition 37 due to the operation $\mathcal{R}\text{epair}_{\text{db1}}$ according to Lemma 5. Similarly, the entire enumeration of all possible updates obtained from $\mathcal{R}\text{epair}_{\text{db1}}$ is then sufficient for overall completeness when $\mathcal{R}\text{epair}_{\text{db1}}$ is complete with respect to the locally least changing updates according to Lemma 5. □

References

1. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1–39 (2008). https://doi.org/10.1145/1322432.1322433
2. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating metamodelling aspects with graph transformation for efficient visual language definition and model manipulation. In: M. Wermelinger, T. Margaria (eds.) Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004 Barcelona, Spain, March 29–april 2, 2004, Proceedings, Lecture Notes in Computer Science, vol. 2984, pp. 214–228. Springer (2004). https://doi.org/10.1007/978-3-540-24721-0_16
3. Barkowsky, M., Giese, H.: Hybrid search plan generation for generalized graph pattern matching. In: E. Guerra, F. Orejas (eds.) Graph Transformation—12th International Conference, ICGT 2019, Held as Part of STAF 2019, Eindhoven, The Netherlands, July 15-16, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11629, pp. 212–229. Springer (2019). https://doi.org/10.1007/978-3-030-23611-3_13
4. Bergmann, G.: Translating OCL to graph patterns. In: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfrán (eds.) Model-Driven Engineering Languages and Systems—17th International Conference, MODELS 2014, Valencia, Spain, September 28–October 3, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8767, pp. 670–686. Springer (2014). https://doi.org/10.1007/978-3-319-11653-2_41
5. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: Proceedings of the Third International Workshop on Graph and Model Transformations, GRaMoT '08, pp. 25–32. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1402947.1402953
6. Beyhl, T., Blouin, D., Giese, H., Lambers, L.: On the operationalization of graph queries with generalized discrimination networks. In: R. Echahed, M. Minas (eds.) Graph Transformation - 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5–6, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9761, pp. 170–186. Springer (2016). https://doi.org/10.1007/978-3-319-40530-8_11
7. Beyhl, T., Giese, H.: Incremental view maintenance for deductive graph databases using generalized discrimination networks. In: A. Heußner, A. Kissinger, A. Wijs (eds.) Proceedings Second Graphs as Models Workshop, GaM@ETAPS 2016, Eindhoven, The Netherlands, April 2–3, 2016., EPTCS, vol. 231, pp. 57–71 (2016). https://doi.org/10.4204/EPTCS.231.5
8. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient subgraph matching by postponing cartesian products. In: F. Özcan, G. Koutrika, S. Madden (eds.) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26–July 01, 2016, pp. 1199–1214. ACM (2016). https://doi.org/10.1145/2882903.2915236
9. Biermann, E., Ermel, C., Taentzer, G.: Precise semantics of EMF model transformations by graph transformation. In: K. Czarnecki, I. Ober, J. Bruel, A. Uhl, M. Völter (eds.) Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28–October 3, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5301, pp. 53–67. Springer (2008). https://doi.org/10.1007/978-3-540-87875-9_4
10. Boronat, A.: Offline delta-driven model transformation with dependency injection. In: Hähnle and van der Aalst [21], pp. 134–150. https://doi.org/10.1007/978-3-030-16722-6_8
11. Búr, M., Ujhelyi, Z., Horváth, Á., Varró, D.: Local search-based pattern matching features in emf-incquery. In: F. Parisi-Presicce, B. Westfechtel (eds.) Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21–23, 2015. Proceedings, Lecture Notes in Computer Science, vol. 9151, pp. 275–282. Springer (2015). https://doi.org/10.1007/978-3-319-21145-9_18
12. Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: On principles of least change and least surprise for bidirectional transformations. J. Object Technol. **16**(1), 3:1–31 (2017). https://doi.org/10.5381/jot.2017.16.1.a3
13. Cheng, Y., Chen, L., Yuan, Y., Wang, G.: Rule-based graph repairing: Semantic and efficient repairing methods. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris,

France, April 16–19, 2018, pp. 773–784. IEEE Computer Society (2018). https://doi.org/10.1109/ICDE.2018.00075
14. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg [37], pp. 313–400
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Berlin (2006)
16. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: $\mathcal{M}$-adhesive transformation systems with nested application conditions. part 2: Embedding, critical pairs and local confluence. Fundam. Inform. **118**(1–2), 35–63 (2012). https://doi.org/10.3233/FI-2012-705
17. Fritsche, L., Kosiol, J., Schürr, A., Taentzer, G.: Efficient model synchronization by automatically constructed repair processes. In: Hähnle and van der Aalst [21], pp. 116–133. https://doi.org/10.1007/978-3-030-16722-6_7
18. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. ECEASST **18**, (2009). https://doi.org/10.14279/tuj.eceasst.18.268
19. Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009). https://doi.org/10.1017/S0960129508007202
20. Habel, A., Sandmann, C.: Graph repair by graph programs. In: M. Mazzara, I. Ober, G. Salaün (eds.) Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25–29, 2018, Revised Selected Papers, Lecture Notes in Computer Science, vol. 11176, pp. 431–446. Springer (2018). https://doi.org/10.1007/978-3-030-04771-9_31
21. Hähnle, R., van der Aalst, W.M.P. (eds.): Fundamental Approaches to Software Engineering—22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11424. Springer (2019). https://doi.org/10.1007/978-3-030-16722-6
22. Hidaka, S., Tisi, M., Cabot, J., Hu, Z.: Feature-based classification of bidirectional transformation approaches. Softw. Syst. Model. **15**(3), 907–928 (2016). https://doi.org/10.1007/s10270-014-0450-0
23. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. ECEASST **6**, (2007). https://doi.org/10.14279/tuj.eceasst.6.49
24. Huisman, M., Rubin, J. (eds.): Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, Lecture Notes in Computer Science, vol. 10202. Springer (2017). https://doi.org/10.1007/978-3-662-54494-5
25. Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.: An integrated semantics for UML class, object and state diagrams based on graph transformation. In: M.J. Butler, L. Petre, K. Sere (eds.) Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15–18, 2002, Proceedings, Lecture Notes in Computer Science, vol. 2335, pp. 11–28. Springer (2002). https://doi.org/10.1007/3-540-47884-1_2
26. Macedo, N., Cunha, A.: Least-change bidirectional model transformation with QVT-R and ATL. Softw. Syst. Model., pp. 783–810. (2016). https://doi.org/10.1007/s10270-014-0437-x
27. Macedo, N., Tiago, J., Cunha, A.: Systematic literature review of model repair approaches. http://tinyurl.com/hv7eh6h. Accessed: 2018-11-14
28. Macedo, N., Tiago, J., Cunha, A.: A feature-based classification of model repair approaches. IEEE Trans. Softw. Eng. **43**(7), 615–640 (2017). https://doi.org/10.1109/TSE.2016.2620145
29. Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: OCL2AC: automatic translation of OCL constraints to graph constraints and application conditions for transformation rules. In: L. Lambers, J.H. Weber (eds.) Graph Transformation—11th International Conference, ICGT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings, Lecture Notes in Computer Science, vol. 10887, pp. 171–177. Springer (2018). https://doi.org/10.1007/978-3-319-92991-0_11
30. Ohrndorf, M., Pietsch, C., Kelter, U., Kehrer, T.: Revision: a tool for history-based model repair recommendations. In: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (eds.) Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27—June 03, 2018, pp. 105–108. ACM (2018). https://doi.org/10.1145/3183440.3183498
31. OMG: Object Constraint Language (2014). http://www.omg.org/spec/OCL/
32. Orejas, F., Boronat, A., Ehrig, H., Hermann, F., Schölzel, H.: On propagation-based concurrent model synchronization. ECEASST **57** (2013). http://journal.ub.tu-berlin.de/eceasst/article/view/871
33. Orejas, F., Pino, E., Navarro, M., Lambers, L.: Institutions for navigational logics for graphical structures. Theor. Comput. Sci. **741**, 19–24 (2018). https://doi.org/10.1016/j.tcs.2018.02.031
34. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. Sci. Comput. Program. **152**, 38–62 (2018). https://doi.org/10.1016/j.scico.2017.08.006
35. Rensink, A.: Representing first-order logic using graphs. In: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (eds.) Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28–October 2, 2004, Proceedings, Lecture Notes in Computer Science, vol. 3256, pp. 319–335. Springer (2004). https://doi.org/10.1007/978-3-540-30203-2_23
36. Rensink, A., Kleppe, A.: On a graph-based semantics for UML class and object diagrams. ECEASST **10**, (2008). https://doi.org/10.14279/tuj.eceasst.10.153
37. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1. Foundations World Scientific, Singapore (1997)
38. Sandmann, C., Habel, A.: Rule-based graph repair. In: R. Echahed, D. Plump (eds.) Proceedings Tenth International Workshop on Graph Computation Models, GCM@STAF 2019, Eindhoven, The Netherlands, 17th July 2019, EPTCS, vol. 309, pp. 87–104 (2019). https://doi.org/10.4204/EPTCS.309.5
39. Schneider, S., Lambers, L., Orejas, F.: Symbolic model generation for graph properties. In: Huisman and Rubin [24], pp. 226–243. https://doi.org/10.1007/978-3-662-54494-5_13
40. Schneider, S., Lambers, L., Orejas, F.: Automated reasoning for attributed graph properties. Int. J. Softw. Tools Technol. Transf. **20**(6), 705–737 (2018). https://doi.org/10.1007/s10009-018-0496-3
41. Schneider, S., Lambers, L., Orejas, F.: A logic-based incremental approach to graph repair. In: Hähnle and van der Aalst [21], pp. 151–167. https://doi.org/10.1007/978-3-030-16722-6_9
42. Schoenboeck, J., Kusel, A., Etzlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M.: CARE - A constraint-based approach for re-establishing conformance-relationships. In: G. Grossmann, M. Saeki (eds.) Tenth Asia-Pacific Conference on Conceptual Modelling, APCCM 2014, Auckland, New Zealand, January 2014, CRPIT, vol. 154, pp. 19–28. Australian Computer Society (2014). http://crpit.com/abstracts/CRPITV154Schoenboeck.html
43. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair. In: Huisman and Rubin [24], pp. 283–299. https://doi.org/10.1007/978-3-662-54494-5_16

44. The Linked Data Benchmark Council (LDBC): Social network benchmark. https://github.com/ldbc/ldbc_snb_docs. Accessed: 2019-08-27
45. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**(1), 31–42 (1976). https://doi.org/10.1145/321921.321925
46. Ullmann, J.R.: Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. ACM Journal of Experimental Algorithmics **15**, (2010). https://doi.org/10.1145/1671970.1921702
47. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for alloy. In: M. Huchard, C. Kästner, G. Fraser (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018, pp. 577–588. ACM (2018). https://doi.org/10.1145/3238147.3238162