Carl von Ossietzky Universität Oldenburg
Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Departement für Informatik

# A Theory on Graph Generation and Graph Repair with Application to Meta-Modeling

Dissertation zur Erlangung des akademischen Grades eines

**Doktors der Naturwissenschaften**

im Fach Informatik

vorgelegt von

Christian Sandmann, M. Sc.

Oldenburg, 27. Juni 2021

# Prüfungskommision

| | |
|---|---|
| Gutachterin: | Prof. Dr. Annegret Habel |
| Gutachterin: | Prof. Dr. Gabriele Taentzer |

Datum der Disputation: 01. Oktober 2021

# Abstract

In meta-modeling, a general problem is to generate instances of a meta-model conforming to a given constraint. Our approach to model generation is rule-based: translate the structure of the meta-model into a graph grammar, translate constraints in the Object Constraint Language into graph constraints, and integrate the graph constraints into the grammar yielding a graph grammar, to generate graphs satisfying the constraints directly. Given a graph grammar and a graph constraint, a graph grammar is constructed that generates *some/all* graphs of the given grammar that satisfy the constraint. Moreover, for specific graph grammars and constraints, the resulting grammar generates *all* graphs satisfying the constraint.

We investigate the generation of instances by grammar-based repair: with a graph grammar generate an instance, which has to be transformed to an instance satisfying a given constraint. Given a specific constraint, a "repair" program is constructed, i.e., a program such that every application of the program to a graph yields a graph satisfying the constraint. The constructed repair programs are stable (nothing happens, if the constraint is satisfied), maximally preserving (items are preserved whenever possible), and terminating.

A model graph based on the Eclipse Modeling Framework (EMF) is a typed graph satisfying some structural EMF-constraints. Application of the repair results to the EMF-world yields model repair for EMF constraints.

# Zusammenfassung

In der Meta-Modellierung ist ein allgemeines Problem, Instanzen eines Meta-Modells zu erzeugen, die einer gegebenen Bedingung entsprechen. Unser Ansatz zur Modellgenerierung ist regelbasiert: Übersetze die Struktur des Meta-Modells in eine Graphgrammatik, übersetze Bedingungen in der Object Constraint Language in Graphbedingungen und integriere die Graphbedingungen in eine Grammatik, was eine Graphgrammatik ergibt, um die Graphen, die die Bedingungen erfüllen, direkt zu erzeugen. Gegeben sei eine Graphgrammatik und eine Graphbedingung, konstruiere eine Graphgrammatik, die *einige/alle* Graphen der Grammatik generiert, die die Bedingung erfüllen. Für bestimmte Graphgrammatiken und bestimmte Graphbedingungen generiert die Ergebnis-Graphgrammatik *alle* Graphen, die die Bedingung erfüllen.

Wir untersuchen die Generierung von Instanzen mit Grammatik-basierter Reparatur: Generiere mit der Graph Grammatik eine Instanz, die in eine Instanz transformiert werden soll, die eine gegebene Bedinung erfüllt. Gegeben sei eine spezielle Bedingung, dann wird ein "Reparatur" Programm extrahiert, das heißt, ein Programm, sodass jede Anwendung des Programms auf einen Graphen einen Graphen ergibt, der die Bedingung erfüllt. Die Reparatur Programme sind stabil (nichts passiert, wenn die Bedingung erfüllt ist), maximal bewahrend (Elemente werden nach Möglichkeit bewahrt) und terminierend.

Ein auf dem Eclipse Modeling Framework (EMF) basierender Modellgraph ist ein getypter Graph, der einige strukturelle EMF Bedingungen erfüllt. Die Anwendung der Ergebnisse zur Reparatur auf die EMF-Welt ergibt Modell Reparatur für EMF Bedingungen.

# Acknowledgements

My work was mainly supported by the German Research Foundation (DFG), Grants HA 2936/4-2 and TA 2941/3-2 (Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages) directed by my supervisors Annegret Habel and Gabriele Taentzer.

Writing this thesis was a challenge I could not have mastered without the help of many people, who provided me guidance and support. First and foremost, I thank my supervisor Annegret Habel for her continuous support and guidance during my time as a Ph.D. student. Her knowledge, advice and support advanced my research skills significantly. Another thank you goes to my second supervisor Gabriele Taentzer for a lot of discussions, a lot of comments to my thesis and helpful feedback. I also want to thank the members of the PhD committee Ernst-Rüdiger Olderog and Maike Schwammberger for many interesting questions on my topic.

My gratitude also goes to the people, who provided me helpful comments and advice. First, I want to thank all my colleagues from the theoretical computer science group, for the pleasant working atmosphere. In particular, Marius Hubatschek, who worked with me on the implementation of the repair concepts and for discussing a lot on the implementation, Okan Oekzan, Berthold Hofmann (Uni Bremen), and the colleagues Jens Kosiol and Nebras Nassar (Uni Marburg) for their helpful comments to my papers and the PhD thesis. Finally, I want to thank Sarah Blum and Okan Oekzan for proofreading various parts of my thesis.

# Contents

# Chapter 1

# Introduction

Graphs and graph transformation are an intuitive and mathematical precise formalism in computer science. Graphs are a natural way of describing complex situations on an intuitive level. The states of a system are modelled as graphs. Moreover, the transformations of graphs into other graphs bring dynamic behaviour into such a system. This yields a wide area of applications of graph transformation systems in computer science, such as graph parsing (Drewes et al. [DHM20]), database design (Bergmann et al. [BHH12], Varró et al. [VFV06]), in the design of concurrent and distributed systems, and software engineering (Heckel and Taentzer [HT20]).

Model-driven engineering (MDE) (Sendall et al. [SK03]) is a development process where the primary artefacts are models. A model is an abstract representation of a system. MDE has gained increased popularity in various engineering disciplines and is used in many industrially relevant engineering disciplines, such as automotive and aerospace domains as well as software development to capture the structure and behaviour of complex systems.

An essential topic of model-driven engineering is the generation of consistent models of meta-models. A *meta-model* is a model specifying the abstract syntax of modeling languages. The consistency of a model may be specified through a set of constraints, which specify, for example, structural properties on the model. A model is said to be consistent if it satisfies the given set of constraints. Since software systems become larger and more complex and models are modified by different stakeholders, it is very likely that inconsistencies arise during the development process.

Structural properties of models have been commonly specified with textual logical formulas. Another alternative is an approach where properties are specified as graph constraints [HW95, HHT96, HP09]. Graph constraints may be seen as a tree of graph mappings equipped with logical quantifiers. This has the advantage that they provide an intuitive and mathematical precise formalism to describe the properties. Many correctness problems for conditions have already been solved, the construction of weakest liberal preconditions, to ensure, that a system is correct under any application of a program. Additionally, it comes with constructions, which preserve or guarantee the correctness of a system under the application of rules changing the system states.

In software engineering, a common approach is to use graphs to represent the structure and behaviour of a system, for example, software architectures, class and object structures, and control flow (Heckel and Taentzer [HT20]). This has the advantage that the mathematical precision of graphs can be used for the formal verification, and the intuitive level provided by graphs is excellent for the visual modeling.

In Taentzer [Tae12], an approach for the translation of the structure of the meta-model into a rule-based system, a so-called typed graph grammar, has been presented. The meta-model is a type graph and the models, i.e., the instances of a meta-model, are represented by graphs that are typed over the type graph. Furthermore, in Radke et al. [RAB$^+$18], an approach for the translation of constraints in the Object Constraint Language (OCL) into graph constraints has been presented. The graph constraints are typed over the meta-model and it is shown that a model satisfies an "Essential" OCL invariant iff its typed graph satisfies the corresponding graph constraint.

In this thesis, we investigate how the rule-based approach can be used for the generation of instances *conforming* to the meta-model and the repair of inconsistent instances into instances *conforming* to the meta-model. In more detail, the first step is to *translate* the structure and the typing of the meta-model into a typed graph grammar as described in Taentzer [Tae12]. Additionally, we *translate* OCL constraints from the meta-model into graph constraints as described in Radke et al [RAB$^+$18].

Following this rule-based approach, we investigate two main ideas:

(1) **Generation.** *Integrate* a graph constraint into a graph grammar yielding a graph grammar generating the graphs satisfying the graph constraints. The resulting graph grammar, equipped with application conditions, is able to **generate** instances conforming to the meta-model directly.

(2) **Repair & grammar-based repair.** With the original graph grammar, *generate* instances, which may not satisfy the constraints at some time (drawn in red in Figure 1.1). From the graph constraints, *construct* a graph program directly from the graph constraint and *apply* the graph program to the instance. Moreover, if the resulting instance shall belong to the language of the grammar, we use the rules of the graph grammar, construct a grammar-based graph pro-

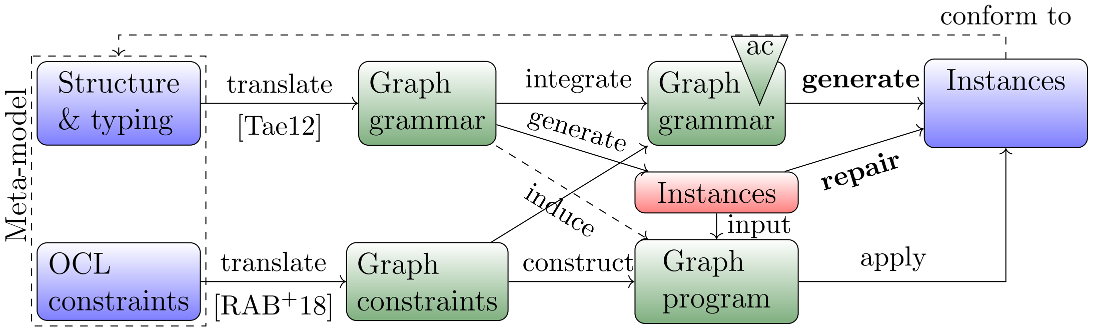


Figure 1.1: Instance generation and repair

gram, and apply it to the instance. This way, **repair** the instances to instances conforming to the meta-model.

We illustrate the problem with Petri nets as the modeling language. We consider a (typed) graph grammar for generating Petri nets and graph constraints for restricting Petri nets. The example is a simplification of the one in Radke et al. [RAB+18] for typed attributed graphs.

**Example 1.** Consider, as an example, a meta-model for modeling a Petri net. The Petri-net grammar is given below.

| | | | |
|---|---|---|---|
| Start graph | $S$ | $=$ | $\boxed{\text{PN}}$ |
| Insert a new place: | `AddPl` | $=$ | $\boxed{\text{PN}} \Rightarrow \boxed{\text{PN}} \xrightarrow{\text{place}} \boxed{\text{Pl}}$ |
| Insert a new transition: | `AddTra` | $=$ | $\boxed{\text{PN}} \Rightarrow \boxed{\text{PN}} \xrightarrow{\text{trans}} \boxed{\text{Tr}}$ |
| Insert a token in a given place: | `AddTok` | $=$ | $\boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$ |
| Connect a transition to a place: | `AddTP` | $=$ | $\boxed{\text{Tr}} \quad \boxed{\text{Pl}} \Rightarrow \boxed{\text{Tr}} \xleftrightarrow[\text{src}]{\text{out}} \boxed{\text{TPArc}} \xleftrightarrow[\text{in}]{\text{tgt}} \boxed{\text{Pl}}$ |
| Connect a place to a transition: | `AddPT` | $=$ | $\boxed{\text{Pl}} \quad \boxed{\text{Tr}} \Rightarrow \boxed{\text{Pl}} \xleftrightarrow[\text{out}]{\text{src}} \boxed{\text{PTArc}} \xleftrightarrow[\text{in}]{\text{tgt}} \boxed{\text{Tr}}$ |

Additionally, the user wants to have a Petri net (PN)

1. consisting of at least one transition (Tr),
2. and, for each place, there exists a token (Tk),

The constraints of the user have been specified as a set of graph constraints, displayed on the left-hand side of each screen in Figure 1.2.

**Generation.** The instance, presented to the user on the right-hand side of the left screen, consists of only a Petri-net node. The instance does not satisfy the first constraint because a transition is missing. The second constraint is satisfied because there is not an occurrence of a place. The approach to instance generation adds the missing transition directly.

The right screen of Figure 1.2 shows a result: the Petri net and the transition. Consequently, all constraints are satisfied.

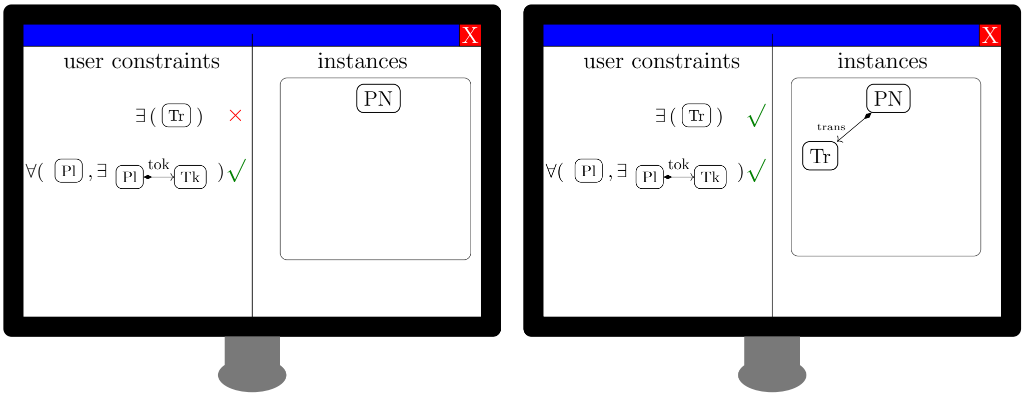


Figure 1.2: Instance generation

**Repair.** Violations of the constraints may arise. Consider, as an example, the illustration in Figure 1.3. By the rules of the graph grammar, a new violation of the second constraint is introduced: there is an occurrence of a place, but the place does not have a token.

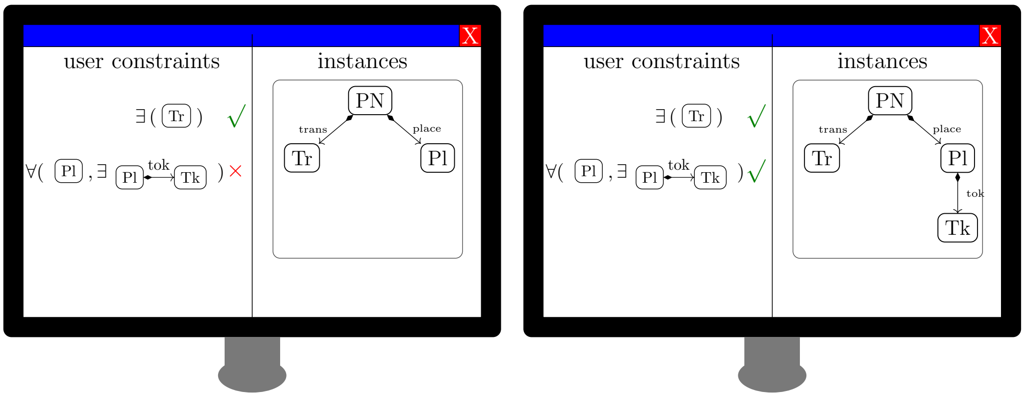


Figure 1.3: Instance repair

This violation can be resolved by adding a new token. The violation of the second constraint has been successfully resolved, the instance satisfies the first constraint.

This repair process has preserved the first constraint, i.e., the transition is still existent. Consequently, this has yielded an instance, which satisfies all the constraints. The result is on the right screen in Figure 1.3.

**State of the art.** The concepts of graph transformation and graph grammars have been studied since the early 1970s (Ehrig et al. [EPS73]). The main idea was to generalize well-known rewriting techniques from strings to graphs. This has yielded to the two main concepts: the double-pushout approach (Ehrig et al. [EEPT06b]) and the single-pushout approach (Löwe [Löw93]). This has been generalized to graph programs, which are minimal and computationally complete (Habel and Plump [HP01]). They have been modified to allow more control mechanisms such as the handing over transformation steps (Pennemann [Pen09]). To describe consistency properties on graphs, graph conditions have been introduced. They are expressively equivalent to first-order constraints on graphs. Additionally, the basic transformations, which guarantee the correctness of rule applications have been presented (Habel and Pennemann [HP09]).

## Contributions

In this thesis, we develop a new theory on graph generation and graph repair (see Figure 1.4). To do so, we extend a theory on graph conditions and a theory on graph programs, to a theory on graph generation and graph repair. We show that, for arbitrary graph grammars and arbitrary constraints, there is an integration of graph constraints into graph grammars using a backward construction. For specific graph grammars and specific constraints, the backward construction terminates. For specific graph constraints, we construct a graph program, such that the application to any (typed) graph yields a (typed) graph satisfying the (typed) graph constraint. Additionally, we prove some properties of the programs, needed for the application to meta-modeling. A model graph based on the Eclipse Modeling Framework (EMF), short EMF-model graph, is a typed graph satisfying some structural EMF-constraints. Application of the results to the EMF-world yields model-repair programs for EMF$k$ constraints, a first-order variant of EMF constraints; application to any typed graph yields an EMF$k$ model graph. Moreover, there is a model-repair for EMF constraints, i.e., for each input graph, there is a program such that the application to any typed graph yields a graph satisfying the EMF constraints. Figure 1.4 summarizes the contributions of this thesis.

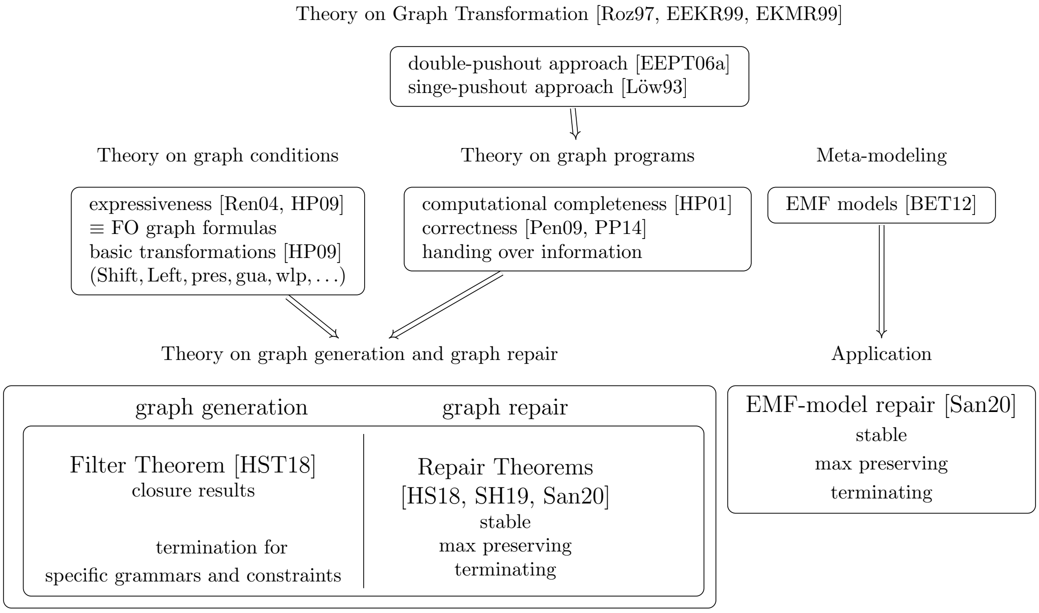

Theory on Graph Transformation [Roz97, EEKR99, EKMR99]
double-pushout approach [EEPT06a]
singe-pushout approach [Löw93]
Theory on graph conditions
Theory on graph programs
Meta-modeling
expressiveness [Ren04, HP09]
≡ FO graph formulas
basic transformations [HP09]
(Shift, Left, pres, gua, wlp, ...)
computational completeness [HP01]
correctness [Pen09, PP14]
handing over information
EMF models [BET12]
Theory on graph generation and graph repair
Application
graph generation
graph repair
Filter Theorem [HST18]
closure results
termination for
specific grammars and constraints
Repair Theorems
[HS18, SH19, San20]
stable
max preserving
terminating
EMF-model repair [San20]
stable
max preserving
terminating

Figure 1.4: Overview of this thesis

# Thesis structure

In Chapter 2, we recall the notions of typed graphs (with containment), graph conditions, rules, grammars and transformations, graph programs with interface, and the basic transformations. In Chapter 3, we investigate the integration of graph constraints into graph grammar and consider the filter problem: Given a graph grammar and a graph constraint, does there exist a grammar that generates all graphs of the original graph language satisfying the constraint. We solve the filter problem for specific graph grammars and specific graph constraints. In Chapter 4, we consider the problem of graph repair: Given a specific graph constraint, does there exist a graph program that "repairs" each input graph, i.e., such that the application to any graph yields a graph satisfying the graph constraint. Additionally, we consider properties of the programs. In Chapter 5, we apply the results on graph repair to the EMF-world and yield model repair programs for EMF$k$ constraints, a first-order variant of EMF constraints; application to any typed graph yields an EMF$k$ model graph. From these results, we derive results for EMF model repair. In Chapter 6, we describe an implementation of the graph repair approach and its usage in ENFORCE$^+$. In Chapter 7, we summarize the results in this thesis and mention some further work. In Appendix A, we provide some details on category theory.

## List of publications

In the following, we list the papers which were published during the doctoral project by the author (in chronological order).

- Annegret Habel, Christian Sandmann, and Tilman Teusch. Integration of graph constraints into graph grammars. In *Graph Transformation, Specifications, and Nets*, volume 10800 of *LNCS*, pages 19 – 36, 2018. $\Longrightarrow$ Chapter 3 is based on this paper.

- Annegret Habel and Christian Sandmann. Graph repair by graph programs. In *Graph Computation Models (GCM 2018)*, volume 11176 of *LNCS*, pages 431 – 446, 2018. $\Longrightarrow$ Chapter 4 is based on this paper.

- Christian Sandmann and Annegret Habel. Rule-based graph repair. In *Graph Computation Models (GCM 2019)*, volume 309 of *EPTCS*, pages 87 – 104, 2019. $\Longrightarrow$ Chapter 4 and Section 4.7 is based on this paper.

- Christian Sandmann. Graph repair and its application to meta-modeling. In *Graph Computation Models (GCM 2020)*, volume 330 of *EPTCS*, pages 13 – 34, Open Publishing Association, 2020. $\Longrightarrow$ Chapter 5 and Section 4.4 is based on this paper.

- Marius Hubatschek and Christian Sandmann. Implementing graph repair in ENFORCE$^+$. Technical Report, University of Oldenburg, `https://uol.de/f/2/dept/informatik/ag/fs/fs-pub/Repair2020_11_25.pdf`, 2020. $\Longrightarrow$ Chapter 6 is based on this paper.

# Chapter 2

# Preliminaries

In this chapter, we recall the notions of typed graphs (with containment), graph conditions, rules and transformations, graph programs, and basic transformations [BET12, HP09, Pen09]. Our concepts are based on Biermann et al. [BET12]. They introduce type graphs with inheritance (and containment).[1] For simplifying the concepts, we ignore the inheritance. It would be interesting to extend our theory to typed graphs with containment and inheritance (see further topics).

We assume that the reader is familiar with the basic notions of graph transformation and the basic concepts of category theory. For a detailed introduction, we refer the reader to [Ehr79, AM75, AHS90].

**Decision (inheritance).** In Biermann et al. [BET12], type graphs with inheritance (and containment) are introduced. For our application, we only introduce typed graphs (with containment). The reason for this, is that our application to meta-modeling (see Chapter 5) considers the EMF constraints, which are formulated on the set of containment edges.

## 2.1 Typed graphs

In the following, we introduce so-called typed graphs based on the paper by Biermann et al. [BET12]. For our application to meta-modeling in mind, we only use typed graphs with containment. Thus, for simplicity, we restrict to typed graphs with containment.

[1] In a preliminary version of [San20], we took typed graphs with containment and inheritance as basis. By a comment of a referee, we simplified our considerations to typed graphs (with containment).

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

**Definition 1 (graphs & morphisms).** A *(directed) graph* $G = (V_G, E_G, \mathrm{s}_G, \mathrm{t}_G)$ consists of a set $V_G$ of *nodes* and a set $E_G$ of *edges*, as well as source and target functions $\mathrm{s}_G, \mathrm{t}_G \colon E_G \to V_G$. Given graphs $G$ and $H$, a *(graph) morphism* $g \colon G \to H$ consists of total functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources and targets, that is, $g_V \circ \mathrm{s}_G = \mathrm{s}_H \circ g_E$ and $g_V \circ \mathrm{t}_G = \mathrm{t}_H \circ g_E$. The morphism $g$ is *injective* (*surjective*) if $g_{\mathrm{V}}$ and $g_{\mathrm{E}}$ are injective (surjective), and an *isomorphism* if it is injective and surjective. In the latter case, $G$ and $H$ are *isomorphic*, denoted by $G \cong H$. A *partial* morphism $G \rightharpoonup H$ is an inclusion $S \hookrightarrow H$ such that $S \subseteq G$.

**Convention.** In drawing graphs, nodes are drawn as circles and edges as arrows. Arbitrary morphisms are drawn by usual arrows $\to$, injective ones by $\hookrightarrow$. An edge is said to be *real* if it is not a loop.

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

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

**Convention.** In drawing of type graphs, nodes are drawn as black circles and the name or “type” of the nodes is drawn inside. Edges are drawn as arrows and the name or “type” of the edge is drawn beside the arrow. The drawing of a type graph is obtained from the underlying graph by marking every containment edge ( ◆⟶ ) with a black diamond at the source, and adding, for every pair $(e_1, e_2)$ of opposite edges, a bidirectional edge ( ⟷ ) between the source and the target of the first edge with two edge type names, one at each end.

**Example 2.** A type graph for Petri nets is given in Figure 2.1.

The type graph consists of the (“type”) nodes PetriNet (PN), Place (Pl), Transition (Tr), Token (Tk), Place-to-Transition Arcs (PTArc), and Transition-to-Place Arcs (TPArc), written inside the nodes, and the (“type”) edges places, trans, tok, in, out, src, tgt, tparcs and ptarcs. The containment edges from the PetriNet to the Place (Transition, PTArc, and TPArc)-node are marked in the graph by a black diamond ◆ at the source of the edge. The opposite edge relation relates the edges from the PTArc (TPArc)-node to the Place (Transition)-node of type src and the Place (Transition)-node to the PTArc (TPArc)-node of type out.

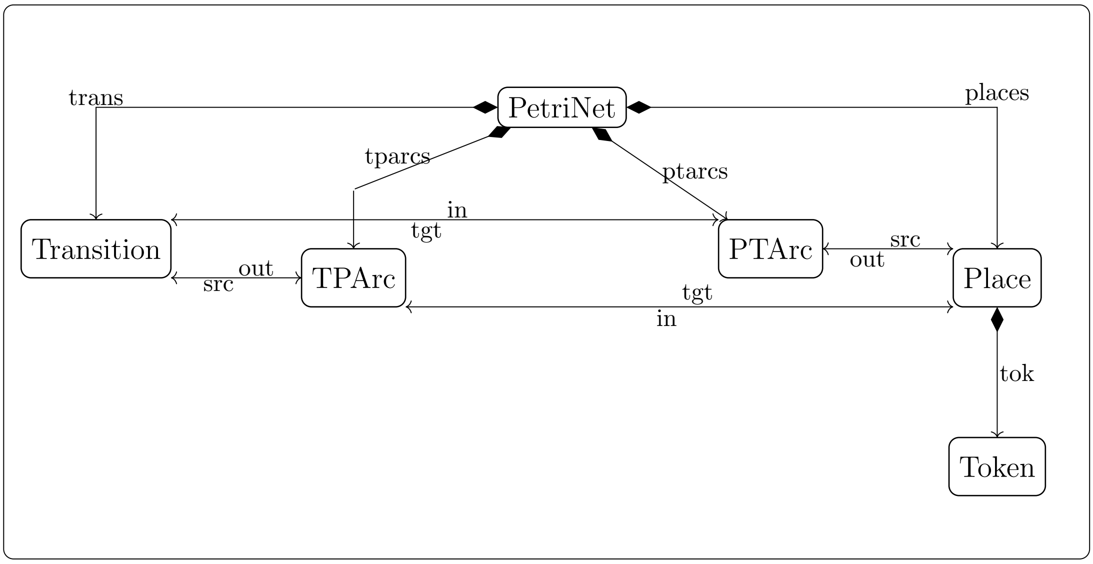


Figure 2.1: Type graph for Petri nets, adapted from [Wac07]

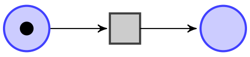
**Assumption 1.** In the following, let $TG = (T, C, O)$ be a fixed type graph.

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

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

$$\begin{tikzcd}
G \arrow[rr, "g"] \arrow[dr, "type_G"'] & {=} & H \arrow[dl, "type_H"] \\
& TG &
\end{tikzcd}$$

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

**Example 3.** The Petri net  consists of two places, and one transition. The left place has exactly one token, the right one has none. One place is connected to the transition via a PTArc, where the place is the source and the transition is the target, and the other place is connected to the transition via a TPArc, where the transition is the source and the place is the target.

The graph $G$ in Figure 2.2 together with the typing morphism *type* is a typed graph, typed over the type graph from Figure 2.1.

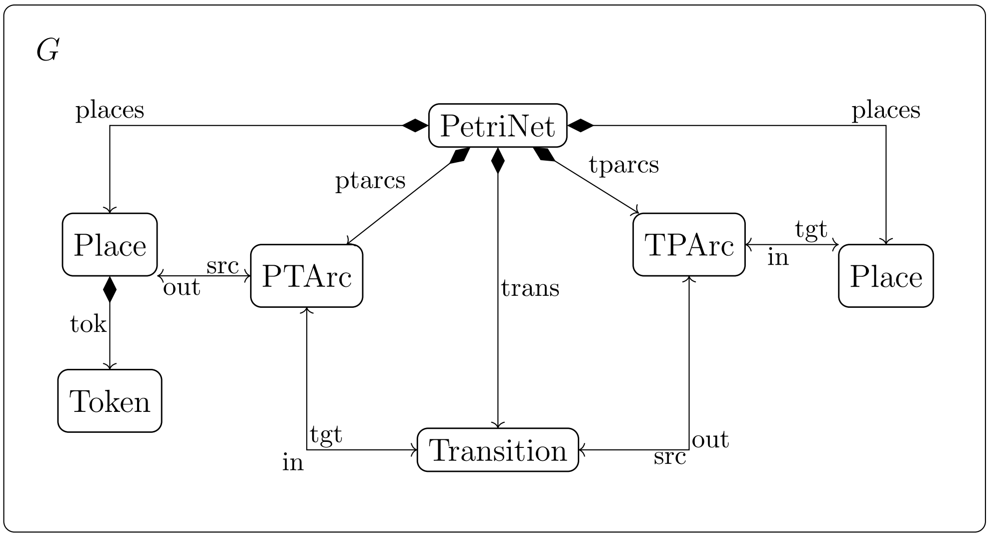


Figure 2.2: Typed graph for Petri nets

The Petri net is related to the typed graph in Figure 2.2 as follows: The typed nodes represent the elements of the Petri net, i.e., a node representing the Petri net, two nodes representing the places, a node representing the Token, and a node representing the Transition. Furthermore, there are nodes representing the edges from the left place to the transition (PTArc) and from the transition to the place (TPArc). The PTArc-node is related with the Place-node by a bidirectional edge typed by src (for source) and out (for outgoing), and with the Transition-node by a bidirectional edge typed by tgt (for target) and in (for incoming). The TPArc is related similar. The Token-node is related with the left Place-ode by a tok-typed edge. Finally, the Place-, Transtion-, PTArc-, and TPArc-node are related with the PetriNet-node by containment edges typed by places, trans, ptarc, and tparc.

Standard graph transformation systems have been studied extensively and applied to several areas of computer science [Roz97, EEKR99, EKMR99]. To cope with the different varieties of graphical structures, they were, first, generalized to high-level replacement (HLR) systems [EHKP91] and then, based on the notion of adhesive categories [LS05], to weak adhesive HLR systems in [EEHP06, EEPT06b] and recently to $\mathcal{M}$-adhesive systems [EGH10], i.e., an $\mathcal{M}$-adhesive category and a set of rules over the category. There is a proper hierarchy of categories: graph ⇒ high-level ⇒ weak adh(esive) HLR ⇒ $\mathcal{M}$-adhesive; categories that show that the implications are proper are given in [EEPT06b, EGH10].

Given a type graph $TG$, all graphs are typed over $TG$ and all morphisms are injective. Typed graphs over $TG$ and injective morphisms form a category $\mathbf{Graphs_{TG}}$. If $\mathcal{M}$ denotes the class of injective morphisms and $\mathcal{E}'$ the class of pairs of jointly surjective morphisms (Ehrig et al. [EEPT06a]), then the category $\mathbf{Graphs_{TG}}$ turns out to be $\mathcal{M}$-adhesive and has a $\mathcal{E}'$-$\mathcal{M}$ pair factorization. As a consequence, we are sure that the category has pushouts (used for the definition of transformations) and there exists a Shift construction (see Lemma 3). The definitions of $\mathcal{M}$-adhesive categories and $\mathcal{E}'$-$\mathcal{M}$ pair factorization can be found in the Appendix A.

The following lemma is the basis for the concepts in the preliminaries.

**Lemma 1.** $\mathbf{Graphs_{TG}}$ is $\mathcal{M}$-adhesive and has $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

**Sketch of proof.** The proof of $\mathcal{M}$-adhesiveness is done by constructing a slice category to the category of typed graphs. By [EEPT06a, Remark 5.26], we show that the category has binary coproducts and $\mathcal{E}'_0$-$\mathcal{M}_0$ pair factorization. From this it follows, that it has $\mathcal{E}'$-$\mathcal{M}$ pair factorization. For the full proof, see the proof in the Appendix A.

**Bibliographic Notes.** There are many graph-like structures in the literature. In Ehrig et al. [EEPT06a], typed graphs (Definition 2.6) over a type graph, attributed graphs (Definition 8.4) over a data signature, and typed attributed graphs (Definition 8.7) over an attributed type graph are introduced. It is shown that these graphs, together with the corresponding morphisms form categories. In [EEPT06a, Chapter 13] and de Lara et al. [dLBE$^{+}$07], the concepts of typed attributed graphs with node type inheritance are introduced. Typed attributed graphs with node type inheritances do not form a category. It is shown that they are fully compatible with the existing concept of typed attributed graph transformation. In Golas et al. [GLEO12], a new category of typed attributed graphs with inheritance is introduced. In Löwe et al. [LKSS15], the inheritance concept is introduced for typed graphs, and it is shown that the considered graphs form a category. In Biermann et al. [BET12], typed graphs with inheritance and containment are introduced. It is shown that a direct transformation step applied at an EMF model graph yields an EMF model graph again. In Köhler et al. [KLT07], graphs with containment edges, and homomorphisms between them are introduced. It is shown that the pushout result in the category of graphs forms a valid pushout in the category of graphs with containment, if and only if the so-called containment condition holds. The *containment condition* intuitively specifies under which conditions a pushout result violates the EMF constraints “At most one container” and “No containment cycle”. For an overview of the different graph-like structures, see Figure 2.3.

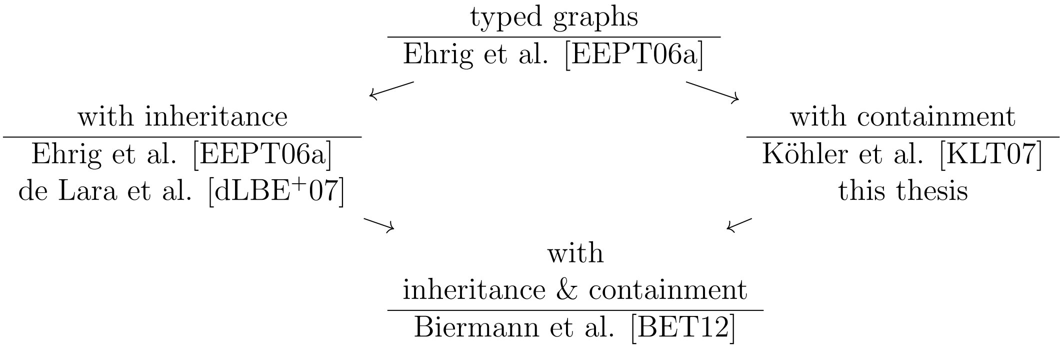


Figure 2.3: Overview of different graph-like structures

## 2.2 Typed graph conditions

In this section, we introduce the concept of typed graph conditions based on the papers by Habel and Pennemann [HP09, Pen09] and present a transformation into a (conjunctive) normal form. For our application, we introduce the concept of conditions with alternating quantifiers. Additionally, we show that every so-called linear condition can be effectively transformed into a condition with alternating quantifiers.

Typed graph conditions allow the formulation of properties for graphs. In particular, a condition can be formulated that a graph must contain (or not contain) a certain subgraph. They are a graphical formalism to specify sets of typed graphs or more formally typed morphisms. Typed graph conditions, also called typed nested graph conditions, are nested constructs that can be represented as trees of morphisms equipped with quantifiers and Boolean connectives. Graph conditions and first-order graph formulas are expressively equivalent.

**Definition 4 (typed graph conditions).** Given a type graph $TG$, a *(typed) (graph) condition*[2] over a graph $A$ is of the form (a) $\texttt{true}$ or $\exists(a, c)$ where $a\colon A \hookrightarrow C$ is a *real* inclusion morphism, i.e., $A \subset C$, and $c$ is a condition over $C$. (b) For a condition $c$ over $A$, $\neg c$ is a condition over $A$. (c) For conditions $c_i$ ($i \in I$ for some finite index set $I$) over $A$, $\wedge_{i\in I} c_i$ is a condition over $A$. Conditions built by (a) and (b) are called *linear*, conditions built by (c) are *conjunctive*. Conditions over the empty graph $\emptyset$ are called *constraints*. In the context of rules, conditions are called *application conditions*. Any morphism $p\colon A \hookrightarrow G$ *satisfies* $\texttt{true}$. A morphism $p$ *satisfies* $\exists\,(a, c)$ with $a\colon\ A \hookrightarrow C$ if there exists an morphism $q\colon C \hookrightarrow G$ such that $q \circ a = p$ and $q$ satisfies $c$.

$$\begin{tikzcd}
 & & TG & & \\
\exists( & A \arrow[ur, hook] \arrow[rr, hook, "a"] \arrow[dr, hook, "p"'] \arrow[rr, phantom, shift right=3, "="] & & C, \arrow[ul, hook'] \arrow[dl, hook', "q"] \arrow[dr, phantom, "\models"] & c \arrow[l, phantom, "\blacktriangleleft"] ) \\
 & & G \arrow[uu, dashed] & &
\end{tikzcd}$$

A morphism $p$ *satisfies* $\neg c$ if $p$ does not satisfy $c$, and $p$ *satisfies* $\wedge_{i\in I} c_i$ if $p$ satisfies each $c_i$ ($i \in I$). We write $p \models c$ if $p$ satisfies the condition $c$ (over $A$). A graph $G$ *satisfies* a constraint $c$, $G \models c$, if the morphism $p\colon \emptyset \hookrightarrow G$ satisfies $c$. A condition $c$ is *satisfiable* if there is a morphism $p$ that satisfies $c$. A constraint $c$ is *satisfiable* if there is a graph $G$ that satisfies $c$. $[\![c]\!]$ denotes the class of all graphs satisfying $c$. Two conditions $c$ and $c'$ over $A$ are *equivalent*, denoted by $c \equiv c'$, if for all graphs $G$ and all morphisms $p\colon A \hookrightarrow G$, $p \models c$ iff $p \models c'$. A condition $c$ *implies* a condition $c'$, denoted by $c \Rightarrow c'$, if for all graphs and all morphisms $p\colon A \hookrightarrow G$, $p \models c$ implies $p \models c'$.

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

**Example 4.** The expression

$$\forall(\emptyset \hookrightarrow \boxed{\text{Pl}}\,, \exists\ \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}\!\bullet\!\!\xrightarrow{\text{tok}}\!\boxed{\text{Tk}}\,, \texttt{true})$$

is a constraint, written in compact form as $\forall(\,\boxed{\text{Pl}}\,, \exists\ \boxed{\text{Pl}}\!\bullet\!\!\xrightarrow{\text{tok}}\!\boxed{\text{Tk}}\,)$ meaning that for each node of type Pl, there exists a real containment edge of type tok to a node

[2] Whenever we have a word in brackets followed by a word, we allow to use the words in combination or neglect the word in brackets, e.g., (graph) conditions stands for graph conditions and, short, conditions.

of type Tk. All graphs are typed over the type graph given in Example 2, where the typing morphism can be unambiguously inferred.

$\forall(\ \boxed{\text{Pl}}\ , \exists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ )$ For all Pl-nodes, there exists a Tk-node and a containment tok-edge.

$\nexists\,(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}}\ )$ There are no two incoming containment tok-edges to a Tk-node.

$\nexists\,(\ \boxed{\text{Pl}} \underset{\text{tok}}{\overset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}}\ )$ There does not exist two tok-edges between a Pl-node and a Tk-node.

**Fact 1 (Expressiveness [HP09]).** Graph conditions, also called first-order graph conditions, are expressively equivalent to first-order graph formulas in the sense of Courcelle [Cou97].

**Decision (first-order graph conditions).** In the following, we consider first-order graph conditions for which a number of basic transformations are known and which have a full graphical representation. Besides first-order graph conditions there are monadic second-order conditions (see the references in the bibliographic notes at the end of this subsection). The monadic second-order conditions in Poskitt and Plump [PP14] have a textual, but not a full graphical representation as first-order conditions. One may restrict on monadic-second order conditions with so-called path conditions. In Lambers and Orejas [OPNL18], they are represented by special edges, i.e., in a graphical way. For path conditions, one would have to extend the basic transformations to this type of conditions.

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

**Definition 5 (alternating quantifiers).** A linear condition of the form

$$\mathrm{Q}(a_1, \overline{\mathrm{Q}}(a_2, \mathrm{Q}(a_3, \ldots))) \text{ with } \mathrm{Q} \in \{\forall, \exists\}, \overline{\forall} = \exists, \overline{\exists} = \forall$$

ending with `true` or `false` is a condition with *alternating quantifiers*.

**Example 5.** The linear conditions $\forall(\ \boxed{\text{Pl}}\ , \exists\,(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ , \texttt{true}\ )\ )$ and $\forall(\ \boxed{\text{Pl}}\ , \exists\,(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ , \texttt{false})\ )$ are conditions with alternating quantifiers.

**Fact 2 (equivalences [Pen09]).** There are the following equivalences:

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

By a conjunctive normal form result for conditions from Pennemann [Pen09], every linear condition effectively[3] can be transformed into an equivalent condition with alternating quantifiers.

**Lemma 2 (alternating quantifiers).** For every linear condition, an equivalent condition with alternating quantifiers can be constructed.

**Proof.** By Pennemann [Pen04], every condition $d$ can be transformed into a normal form, where the following conditions (1) - (3) are satisfied:

(1) Every negation symbol is innermost, i.e., every negated subcondition is of the form $\exists\,a$.

(2) Every Boolean subcondition is in disjunctive (conjunctive) normal form.

(3) There is no subcondition `true` or `false` in $d$, unless $d$ is `true` or `false`.

The condition $d$ is transformed into this normal form by:

(1) Moving $\neg$ inwards.

(2) Transformation in disjunctive normal form.

(3) Elimination of `true` and `false`.

The normal form is transformed into a normal form with alternating quantifiers:

(4) Apply the equivalence $\nexists\,(a,\neg c) \equiv \forall(a,c)$ strictly read from left to right as long as possible.

By the equivalences for conditions, this yields an equivalent condition with alternating quantifiers. □

In Chapter 3, we also use conditions in disjunctive normal form. A condition is in *disjunctive normal form* (DNF) if it is a Boolean formula over positive conditions in disjunctive normal form. In general, for a condition in normal form, there need not exist an equivalent condition in disjunctive normal form, e.g., the condition $\forall(A,\vee_{i\in I}\exists\,C_i)$ is in normal form, but there is no equivalent condition in disjunctive normal form.

[3] In mathematics, an *effective* construction means that it is algorithmically computable.

**Bibliographic notes.** Many formalisms for expressing graph properties as constraints or application conditions have been proposed in the literature. In Orejas and Lambers [OL10], symbolic graph conditions are introduced, which consist of an $E$-graph (see Ehrig et al. [EEPT06a]) whose labels are variables, together with a set of formulas that constraint the possible values of these variables. In Poskitt and Plump [PP14], graph conditions are extended to a monadic-second order logic on graphs. New quantifiers for node- and edge-set variables are introduced and morphisms are equipped with constraints about set membership. In Radke [Rad16], HR* conditions are introduced. They extend graph conditions in the sense that variables may be replaced by a hyperedge replacement system (see Habel [Hab92]). This provides a finite way to express structures of arbitrary size. In Flick [Fli16], conditions are extended by recursive definitions. By Pennemann [Pen09], there is a well-developed theory of graph conditions. This is the reason, we have chosen this setting.

## 2.3 Typed graph programs

In this section, we recall typed rules, typed graph grammars, and typed graph programs with interface based on the thesis of Pennemann [Pen09]. The advantage of programs with interface is the explicit control over marking and unmarking of graph elements.

Typed graphs and injective morphisms form an $\mathcal{M}$-adhesive category (see Lemma 1). Rules are specified by a pair of morphisms, interface morphisms, and an application condition. By the interfaces, it becomes possible to hand over information between the transformation steps. For restricting the applicability of the rule, the rules are equipped with an application condition. An illustration of a rule with interface, application condition, and a transformation is given in Figure 2.4.

**Definition 6 (typed rules & transformations).** Given a category $\mathbf{Graphs_{TG}}$, a *(typed) rule* $\varrho = \langle x, p, \mathrm{ac}, y\rangle$ (*with interfaces* $X$ and $Y$) consists of a plain rule $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$ of morphisms $l\colon K \hookrightarrow L, r\colon K \hookrightarrow R$, with (typed) graphs $L, K$, and $R$, called left-hand side, gluing graph, and right-hand side, respectively, morphisms $x\colon X \hookrightarrow L$, $y\colon Y \hookrightarrow R$, the *(left and right) interface morphisms*, and a left application condition ac over $L$. The partial morphism $i\colon X \rightharpoonup Y$ with $i = y^{-1} \circ r \circ l^{-1} \circ x$ is the *interface morphism* of the rule $\varrho$. If the domain of an interface morphism is empty or the application condition ac is `true`, then the component may not be written.

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

(1) Mark a morphism $g' \colon L \hookrightarrow G$, called *match*, satisfying the dangling condition (see below), such that $g = g' \circ x$ and $g' \models \mathrm{ac}$.

(2) Apply the plain rule $p$ at $g'$ yielding a morphism $h' \colon R \hookrightarrow H$.

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

$$\begin{tikzcd}
 & \overset{\mathrm{ac}}{\triangledown} & TG & & \\
X \arrow[urr, bend left] \arrow[r, hook, "x"] \arrow[dr, "g"'] \arrow[rrrr, dashed, bend right, "i"] \arrow[dr, phantom, shift left=3, "="] & L \arrow[ur] \arrow[d, hook, "g'"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[u] \arrow[d] & R \arrow[ul] \arrow[d, hook, "h'"] & Y \arrow[l, hook', "y"'] \arrow[ull, bend right] \arrow[dl, "h"] \arrow[dl, phantom, shift right=3, "="] \\
 & G & D \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] & H &
\arrow[from=2-2, to=3-3, phantom, "(1)"]
\arrow[from=2-3, to=3-4, phantom, "(2)"]
\end{tikzcd}$$

Figure 2.4: A direct transformation

The application of a plain rule is as in the double-pushout approach [EEPT06b] in the category of typed graphs. A plain rule $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ is applicable to a graph $G$ w.r.t. a morphism $g' \colon L \hookrightarrow G$, iff $g'$ satisfies the *dangling condition*: "No edge in $G - g'(L)$ is incident to a node in $g'(L - K)$."

The *semantics* of the rule $\varrho$ is the set $[\![\varrho]\!]$ of all triples $\langle g, h, i \rangle$ of a morphism $g \colon X \hookrightarrow G$, a morphism $h \colon Y \hookrightarrow H$, and a partial interface morphism $i \colon X \rightharpoonup Y$ with $i = y^{-1} \circ r \circ l^{-1} \circ x$. Instead of $\langle g, h, i \rangle \in [\![\varrho]\!]$, we write $g \Rightarrow_{\varrho,i} h$ or short $g \Rightarrow_{\varrho} h$, or $G \Rightarrow_{\varrho,g,h,i} H$ or short $G \Rightarrow_{\varrho} H$ if the domain of $g$ is empty.

Given graphs $G, H$ and a finite set $\mathcal{R}$ of rules, $G$ *derives* $H$ by $\mathcal{R}$ if $G \cong H$ or there is a sequence of direct transformations $G = G_0 \Rightarrow_{\varrho_1,g_1,h_1} G_1 \Rightarrow_{\varrho_2,h_1,h_2} \ldots \Rightarrow_{\varrho_n,h_{n-1},h_n} G_n \cong H$ with $\varrho_1, \ldots, \varrho_n \in \mathcal{R}$. In this case, we write $G \Rightarrow^*_{\mathcal{R}} H$ or just $G \Rightarrow^* H$. Two transformations $t, t'$ from $G$ to $H$ are *equivalent*, denoted $t \equiv t'$, if for each extension (see Ehrig et al. [EEPT06a]) from $G^*$ to $H^*$ there is an extension of $t'$ from $G^*$ to $H^*$.

**Example 6.** In Figure 2.5, a rule with interface $\boxed{\text{Pl}}$ is shown, typed over the type graph from Example 2. For the graph, the aim is to construct a Petri net where each Pl-node has a tok-edge to a Tk-node. The rule consists of a plain rule, an application condition and interface. The glueing graph consists of the elements, which have to be preserved, i.e., the Pl-node and the Tk-node; the plain rule adds a tok-edge between a Pl-node and Tk-node. The rule is equipped with an application condition $\mathrm{ac} = \nexists(\,\boxed{\text{Pl}}\ \boxed{\text{Tk}} \hookrightarrow \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}\,)$. By the application condition, the rule is applicable if the marked Pl-node does not possess a tok-edge to the Tk-node. The left interface restricts the application to a previously marked place, i.e., the right place in the graph (drawn in sky-blue). By the left interface morphism, the rule can only be applied at that position. The right interface restricts the application of the next rule: By the morphism $h$, the next rule can only be applied at the right place.

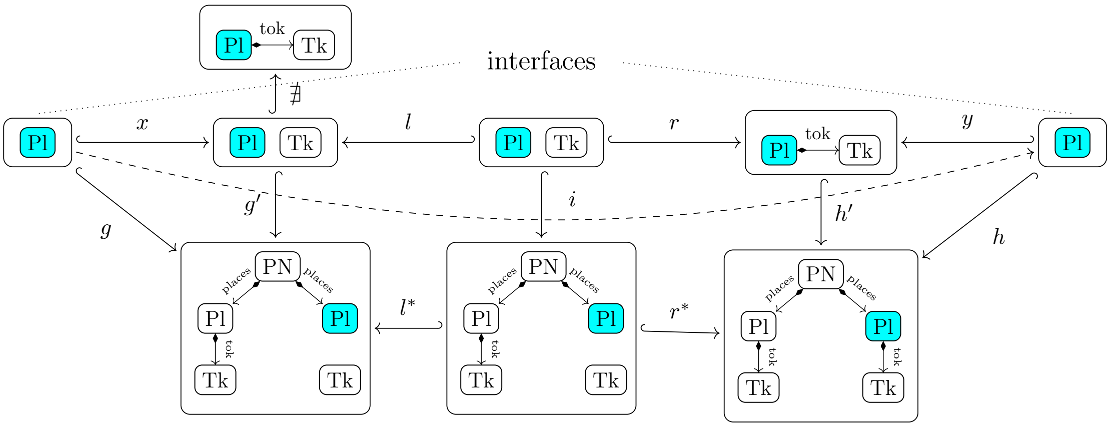


Figure 2.5: A direct transformation

Intuitively, the rule requires a Pl-node and Tk-node, and at a marked Pl-node, i.e., the right-most Pl-node in the graph, the rule adds a containment tok-edge, provided it does not exist one.

In the following, a plain rule $\langle L \hookleftarrow K \hookrightarrow R\rangle$ sometimes is abbreviated by $L \Rrightarrow R$ where the nodes in $L$ and $R$, related by the gluing graph $K$, are indexed with the same index.[4] In a similar way, the match of a rule can be marked by indexing corresponding nodes or using, e.g., the blue color.

**Decision (DPO / SPO).** In the following, we use the single-pushout (SPO) approach by Löwe [Löw93]. Since the SPO approach is not so well-known, we use the double-pushout (DPO) approach plus a so-called *dangling-edges operator*. Given a rule equipped with the dangling-edges operator and a match, first, the dangling edges are deleted such that the dangling condition becomes satisfied, and, afterwards, the rule is applied as in the DPO approach.

[4] Without loss of generality, the gluing graph $K$ is discrete (otherwise, the edges in $K$ are deleted). For rules with discrete gluing graph $K$, the long version $\langle L \hookleftarrow K \hookrightarrow R\rangle$ can be uniquely inferred from the short version).

For node-decreasing rules with plain rule $\langle L \hookleftarrow K \hookrightarrow R\rangle$ satisfying $|V_L| > |V_K|$, the dangling condition may not be satisfied. In this case, $\varrho'$ means that the rule shall be applied in the single-pushout (SPO)-style of replacement [Löw93], i.e., first to remove the dangling edges, and, afterwards application of the rule in the DPO-style. Note that this style of replacement also can be described by a DPO program that fixes a match for the rule, deletes the dangling edges, and afterwards applies the rule at the match. The proceeding can be extended to sets of rules: For a rule set $\mathcal{S}$, $\mathcal{S}' = \{\varrho' \mid \varrho \in \mathcal{S}\}$.

**Example 7.** For the rule $p = \langle\ \boxed{\text{Pl}}\ \ \boxed{\text{Pl}}\ \Rightarrow\ \boxed{\text{Pl}}\ \rangle$, $p'$ denotes that the rule shall be applied in the single-pushout style of replacement, i.e., first, mark an occurrence of the Pl-node and delete the dangling edges, afterwards, apply the rule as in the double-pushout approach.

**Decision (Interface & markings).** Rules with interfaces enable the control over marking and unmarking (see, e.g., Poskitt and Plump [PP13]) of elements in a typed graph and are capable of handling the markings over transformation steps. The left interface restricts the application of the rule to a previously marked context: Given a morphism $g\colon X \hookrightarrow G$, the application is restricted to those morphisms $g'\colon L \hookrightarrow G$ that fit to $g$, i.e., $g = g' \circ x$. The right interface restricts the application of the next rule: By the morphism $h\colon Y \hookrightarrow H$, the next rule can only be applied at $Y$. We use the interface (or marking) to restrict the applicability of a program to a previously marked morphism. Instead of rules with interfaces in the sense of [Pen09], we could use markings as, e.g., in [PP13]. Rules with interfaces may be seen as rules with markings: Whenever there is a marking of a graph $A$ in a graph $G$, i.e., a morphism from $A$ to $G$, choose an extended marking of $C$ in $G$, i.e., a morphism from $C$ to $G$, apply the marked program at that marked position, and, finally, unmark the occurrence. Rules with interfaces may be seen as a formal morphism-based version of the idea of markings combined with rules. We have decided to use the interfaces instead of markings because we have multiple markings and the description of markings by morphisms makes transparent what happens.

Rules may be collected to transformation systems and grammars.

**Definition 7 (graph grammars).** Let $TG = (T, C, O)$ be a type graph. A *graph grammar* $GG = (\mathcal{N}, \mathcal{R}, S)$ over $TG$ consists of a finite set $\mathcal{N} \subseteq T_V \cup T_E$ of nonterminal symbols, a finite set $\mathcal{R}$ of rules over $TG$, and a typed start graph $S$ over $TG$. In the case of $\mathcal{N} = \emptyset$, we write $GG = (\mathcal{R}, S)$ instead of $GG = (\emptyset, \mathcal{R}, S)$.

The *graph language* generated by $GG$ consists of all typed graphs without nonterminal types *derivable* from $S$ by $\mathcal{R}$: $L(GG) = \{G \in \mathcal{G} \mid S \Rightarrow^*_{\mathcal{R}} G\}$ where $\mathcal{G}$ denotes the class of all typed graphs without nonterminal symbols.

Typed graph programs are made of sets of typed rules with interface, non-deterministic choice $\{P, Q\}$, sequential composition $\langle P; Q\rangle$, "as long as possible" iteration $P{\downarrow}$, and the try statement `try` $P$. The definition follows mainly the existing approach of Pennemann [Pen09], and the `try` statement is adapted from Poskitt and Plump [PP13].

**Definition 8 (typed graph programs).** The set of *(typed) (graph) programs with interface* $X$ , $\mathrm{Prog}(X)$, is defined inductively:

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

The statement `Skip` denotes the identity rule $\mathrm{id}_X = \langle X \hookleftarrow X \hookrightarrow X\rangle$. Programs of the form $\langle\langle P_1; P_2\rangle; P_3\rangle$ and $\langle P_1; \langle P_2; P_3\rangle\rangle$ are considered equal and can be written as $\langle P_1; P_2; P_3\rangle$.

Programs with interface transform morphisms. The semantics of a program $P$ consists of triples $\langle g, h, i\rangle$ where the first two morphisms $g$ and $h$ represent the *input* and *result*, respectively, while the last partial morphism is an "interface relation" from the domain of the input to the domain of the result morphism. The input interface represents a number of elements that are assumed to be present in the input graph at the beginning of a computation.

**Definition 9 (semantics).** The *semantics* of a program $P$ with interface $X$, denoted by $[\![P]\!]$, is a set of triples such that, for all $\langle g, h, i\rangle \in [\![P]\!]$, $i \in \mathcal{M}$, the domain of $g$ and $i$ is $X$ and the domain of $h$ is equal to the codomain of $i$:

$$\begin{array}{lll}
(1) & [\![\varrho]\!] & \text{as in Definition 6,} \\
(2) & [\![\{P, Q\}]\!] & = \; [\![P]\!] \cup [\![Q]\!], \\
(3) & [\![\langle P; Q\rangle]\!] & = \; \{\langle g_1, h_2, i_2 \circ i_1\rangle \mid \langle g_1, h_1, i_1\rangle \in [\![P]\!], \langle g_2, h_2, i_2\rangle \in [\![Q]\!],\, h_1 = g_2\}, \\
(4) & [\![\texttt{try}\ P]\!] & = \; \{\langle g, h, i\rangle \mid \langle g, h, i\rangle \in [\![P]\!]\} \cup \{\langle g, g, \mathrm{id}\rangle \mid \nexists\, h. \langle g, h, i\rangle \in [\![P]\!]\}, \\
 & [\![P\downarrow]\!] & = \; \{\langle g, h, \mathrm{id}\rangle \in P^* \mid \nexists h'. \langle h, h', \mathrm{id}\rangle \in [\![\texttt{Fix}(P)]\!]\},
\end{array}$$

where $P^* = \bigcup_{j=0}^{\infty} P^j$ with $P^0 = \texttt{Skip}$, $P^j = \langle \texttt{Fix}(P); P^{j-1}\rangle$ for $j > 0$ and $[\![\texttt{Fix}(P)]\!] = \{\langle g, h \circ i, \mathrm{id}\rangle \mid \langle g, h, i\rangle \in [\![P]\!], i \in \mathcal{M}\}$. Two programs $P, P'$ are *equivalent*, denoted $P \equiv P'$, if $[\![P]\!] = [\![P']\!]$. Instead of $\langle g, h, i\rangle \in [\![P]\!]$, we write $g \Rightarrow_{P,i} h$ or short $g \Rightarrow_P h$ and $G \Rightarrow_{P,g,h,i} H$ or short $G \Rightarrow_P H$ if the domain of $g$ is empty. A program is *terminating*, if the relation $\Rightarrow$ is terminating.

The semantics of the sequential composition implies that a program with interface $X$ may only be iterated if the output interface of the previous computation equals the (input) interface $X$. The iteration as long as possible depends on the reflexive, transitive closure of a program. The statement `Fix` is a generic way of making programs iteratable that do not delete or unmark elements of their interface. It ensures that every possible computation ends with the output interface $X$ by finally unmarking all elements additionally marked during a run of the program. The semantics of `Fix` corresponds to a specific `Unmark` statement at the end of each program branch. The try statement `try` $P$ depends on the "successful" application of the program $P$. If the application of $P$ yields a result, this result is kept. Otherwise, if the program does not yield a result, the changes of $P$ are discarded.

**Example 8.** The program `try` $\varrho$, where the rule

$$\varrho = \langle\ x,\ \boxed{\text{Pl}}\ \boxed{\text{Tk}}\ \Rightarrow\ \boxed{\text{Pl}}\!\bullet\!\!\xrightarrow{\text{tok}}\!\boxed{\text{Tk}}\ ,\nexists\ \boxed{\text{Pl}}\!\bullet\!\!\xrightarrow{\text{tok}}\!\boxed{\text{Tk}}\ , y\ \rangle$$

is the rule from Example 6, is a program with interface. It adds a tok-edge, provided that there is no containment tok-edge from the Pl-node to the Tk-node. If the rule is not applicable, the input morphism is returned.

The program

$$P = \langle\ \boxed{\text{Pl}}\overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}}\boxed{\text{Tk}}\ \Rightarrow\ \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}\ \rangle\!\downarrow$$

deletes one of the tok-edges between a Pl-node and a Tk-node as long as possible.

**Bibliographic notes.** In Habel and Plump [HP01], a programming language based on the nondeterministic application of a set of graph transformation rules, sequential composition, and iteration is introduced. It is shown that this language is minimal in the sense that neither the sequential composition nor the iteration can be omitted to compute every computable partial function on labelled graphs. In Pennemann [Pen09], programs with interface are introduced. The programs with interface are based on four constructs, that is, the selection, deletion, addition, and unselection of elements. The main extension is the explicit control over the selection and unselection of graph elements and the capability of handing over this information between computation steps. It is shown that each program in the sense of Habel and Plump can be seen as a program with the empty graph as interface, i.e., $\langle G, H\rangle \in [\![P]\!]_{\text{HP01}}$ if and only if $\langle i_G, i_H, \text{id}_I\rangle \in [\![P]\!]$, where $i_G, i_H$ denote the morphism from the empty graph to the graph $G, H$, respectively, $I$ is the empty interface, and $[\![P]\!]_{\text{HP01}}$ is the input and output semantics, defined as in Habel and Plump [HP01]. In Poskitt and Plump [PP14], the language of Habel and Plump has been extended, most significantly with conditional branching, i.e., the if-then-else and try-then-else statements. The semantics of `try` in Poskitt and Plump [PP13] and this thesis coincide. In this thesis, the programs with interface of Pennemann [Pen09] are extended by the try statement as in Poskitt and Plump [PP13].

In [EEdL$^+$05, Tae12], layered rule sets are introduced that possess a rule layer function $rl\colon \mathcal{R} \to \mathbb{N}$. The generated language consists of all graphs that can be obtained from a given start graph by applying the rules in the first layer as first, the rules in the second layer as second, and so on. Since the operator "arbitrary often" can be simulated with the help of the operator "as long as possible", a layered rule set may be seen as a special graph program in the sense of Habel and Plump [HP01].

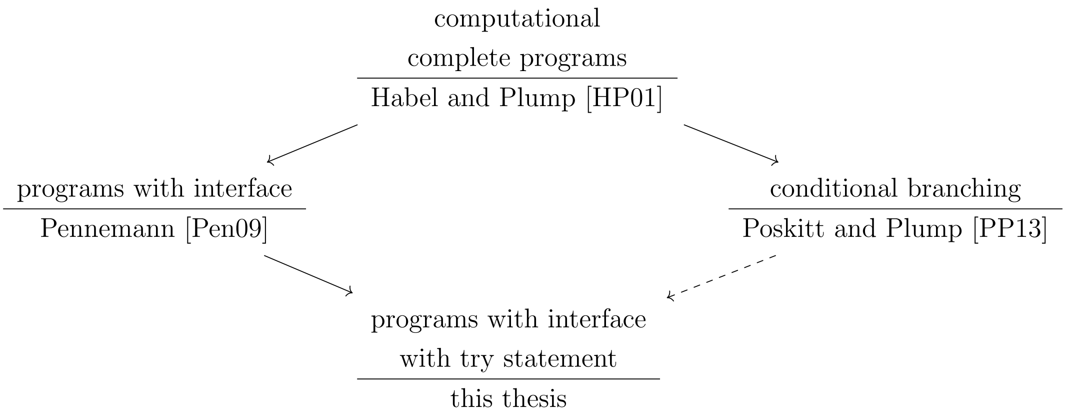


Figure 2.6: Overview of graph programs

## 2.4 Basic transformations

In the following, we consider a number of transformations of conditions, summarized as basic transformations [HP09]. The most important transformations are the shift of conditions over morphisms and rules (see Lemma 3 and 4). Moreover, there are transformations of rules into condition-guaranteeing or -preserving rules. All these transformations can be done because the category of typed graphs is $\mathcal{M}$-adhesive and has an $\mathcal{E}'$-$\mathcal{M}$ pair factorization (Lemma 1).

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

**Lemma 3** **(**Shift **[HP09]).** There is a construction Shift, such that for each condition $d$ over $P$ and every morphism $b\colon P \hookrightarrow R, n\colon R \hookrightarrow H$,

$$n \circ b \models d \iff n \models \mathrm{Shift}(b, d).$$

$$\begin{tikzcd}
d \arrow[r, phantom, "\triangleright"] & P \arrow[rr, hook, "b"] \arrow[dr, hook, "n \circ b"'] & \arrow[d, phantom, "="] & R \arrow[r, phantom, "\triangleleft"] \arrow[dl, hook', "n"] & \mathrm{Shift}(b, d) \\
& & H & &
\end{tikzcd}$$

The relationship between the condition $d$ and the condition $\mathrm{Shift}(b, d)$ is depicted above: The gray triangles left and right to the graph $P$ and $R$, respectively, indicate the conditions $d$ over $P$ and $\mathrm{Shift}(b, d)$ over $R$. The combined morphism $n \circ b$ satisfies the condition $d$ iff the morphism $n$ satisfies the condition $\mathrm{Shift}(b, d)$, the condition $d$ shifted over the morphism $b$.

**Construction 1.** The construction Shift is as follows.

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

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

where a pair $(a', b')$ is *jointly surjective* if for each $x \in R'$ there is a preimage $y \in R$ with $a'(y) = x$ or $z \in C$ with $b'(z) = x$.

**Example 9.** Consider the condition

$$d = \nexists\,(\ \underset{1}{\boxed{\mathrm{Pl}}} \hookrightarrow \underset{1}{\boxed{\mathrm{Pl}}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}}\ )$$

and the morphism $y\colon \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}$. Shifting the condition $\nexists\, a$ over the morphism $y$ yields the application condition $\mathrm{ac}_R = \mathrm{Shift}(y, d) = \mathrm{Shift}(y, \neg\exists\, a) = \neg\mathrm{Shift}(y, \exists\, a) = \neg(\bigvee_{i=1}^{3} \exists\,(a'_i, \mathrm{Shift}(b'_i, \mathtt{true}))) = \neg(\bigvee_{i=1}^{3} \exists\,(a'_i, \mathtt{true})) = \bigwedge_{i=1}^{3} \nexists\,(a'_i)$. where the morphisms $a'_1, a'_2$, and $a'_3$ are in Figure 2.7. In the following, for simplicity, we only the condition $\nexists\, a'_3$ and indicate the other conditions ($\nexists\, a'_1$ and $\nexists\, a'_2$) by dots. For the morphism $x\colon \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}}\ \boxed{\mathrm{Tk}}$ and the condition $d$, the construction $\mathrm{Shift}(x, d)$ yields the application condition $\nexists\,(\ \underset{1}{\boxed{\mathrm{Pl}}} \xrightarrow{\mathrm{tok}} \underset{2}{\boxed{\mathrm{Tk}}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}}\ ) \wedge \ldots$.

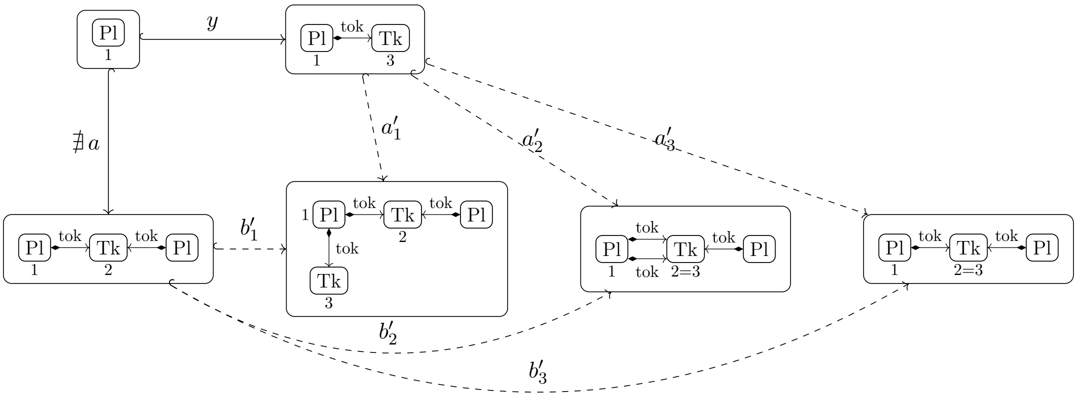


Figure 2.7: Shift of the condition $\nexists\, a$ over the morphism $y$

The construction Left "shifts" a right application condition over a rule into a left application condition.

**Lemma 4 (**Left **[HP09]).** There is a construction Left, such that for each rule $\varrho$ with plain rule $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ and each condition ac over $R$, for each $G \Rightarrow_{\varrho,g,h} H$, $g \models \text{Left}(\varrho, \text{ac}) \iff h \models \text{ac}$.

**Construction 2.** The construction Left is as follows.

$$\begin{tikzcd}
R \arrow[d, "a"'] \arrow[r, phantom, "(1)", shift right=6ex] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[r, phantom, "(2)", shift right=6ex] & L \arrow[d, "a'"] \\
R' & K' \arrow[l, hook'] \arrow[r, hook] & L' \\
\blacktriangle & & \blacktriangle \\
\text{ac} & &
\end{tikzcd}$$

Left($\varrho$, `true`) := `true`.
Left$(\varrho, \exists\,(a, \text{ac})) := \exists\,(a', \text{Left}(p', \text{ac}))$ if $p^{-1}$ is applicable w.r.t. the morphism $a$, $p' := \langle L' \hookleftarrow K' \hookrightarrow R' \rangle$ is the *derived* rule, and `false`, otherwise.
Left$(\varrho, \neg\text{ac})$:=$\neg$Left$(\varrho, \text{ac})$.
Left$(\varrho, \wedge_{i \in I}\text{ac}_i) := \wedge_{i \in I}\text{Left}(\varrho, \text{ac}_i)$.

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

**Example 10.** Consider the plain rule $p = \langle\ \boxed{\text{Pl}}\ \ \boxed{\text{Tk}} \hookleftarrow \boxed{\text{Pl}}\ \ \boxed{\text{Tk}} \hookrightarrow \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}\ \rangle$ and the application condition $\text{ac}_R = \nexists\,(\ \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}\xleftarrow{\text{tok}}\boxed{\text{Pl}}\ ) \wedge \ldots$ from Example 9. The construction Left yields the application condition $\text{ac}_L = \text{Left}(p, ac_R) = \nexists\,(\ \boxed{\text{Pl}}\qquad \boxed{\text{Tk}}\xleftarrow{\text{tok}}\boxed{\text{Pl}}\ ) \wedge \ldots$ over the left-hand side of the rule $p$.

Conditions can be integrated into left application conditions of a rule such that every transformation is "condition-preserving". The statement on Pres, in Habel and Pennemann [HP09], is proven for the case that the domain of the morphism is the empty graph. It can be generalized to rules with left and right interface morphisms and conditions over the same domain as the morphisms.

**Idea.** The construction of condition-preserving rules shifts the condition $d$ over the left interface morphism to the left-hand side of the rule (see Figure below). Furthermore, the condition is shifted over the right interface morphism $y$ to the right-hand side of the rule, the resulting right application condition is shifted over the rule to the left-hand side. The implications of the conditions yield the condition-preserving application condition. An illustration is given in Figure 2.8.

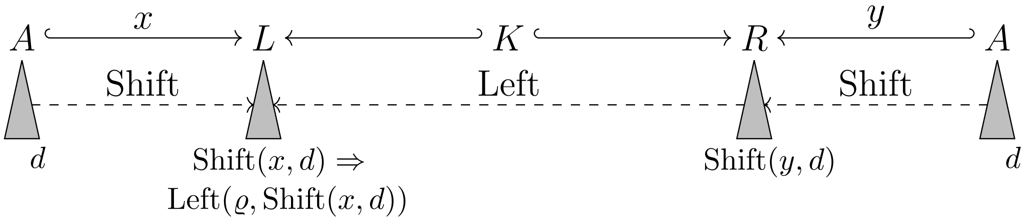

$A$
$x$
$L$
$K$
$R$
$y$
$A$
Shift
Left
Shift
$d$
$\mathrm{Shift}(x,d) \Rightarrow$
$\mathrm{Left}(\varrho, \mathrm{Shift}(x,d))$
$\mathrm{Shift}(y,d)$
$d$

Figure 2.8: Illustration of condition-preserving application conditions

**Lemma 5** (Pres)**.** For each rule $\varrho$ with interfaces $A$ and each condition $d$ over $A$, a condition $\mathrm{ac}_{pres} = \mathrm{Pres}(\varrho, d)$ can be constructed such that all transformations $g \Rightarrow_{\langle \varrho, \mathrm{ac} \rangle} h$ are *d-preserving*, i.e., $g \models d$ implies $h \models d$.

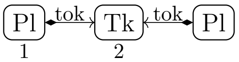
**Construction 3.** $\mathrm{Pres}(\varrho, d) := \mathrm{Shift}(x,d) \Rightarrow \mathrm{Left}(\varrho, \mathrm{Shift}(y,d))$.

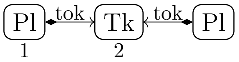
**Proof.** For every transformation $g \Rightarrow_{\langle \varrho, \mathrm{Pres}(\varrho,d) \rangle} h$, ($g \models d$ impl. $h \models d$) iff ($g \models d$ impl. $h' \models \mathrm{Shift}(y,d)$) iff ($g \models d$ impl. $g' \models \mathrm{Left}(p, \mathrm{Shift}(y,d))$) iff ($g' \models \mathrm{Shift}(x,d)$ impl. $g' \models \mathrm{Left}(p, \mathrm{Shift}(y,d))$) iff $g' \models \mathrm{Pres}(\varrho, d)$. □

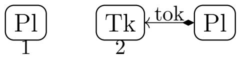
**Example 11.** Given a rule $\varrho = \langle\ x, \boxed{\mathrm{Pl}}\ \boxed{\mathrm{Tk}} \Rightarrow \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}, y\ \rangle$ with interface morphisms $x\colon \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}}\ \boxed{\mathrm{Tk}}$, $y\colon \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}$, and a condition

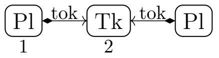
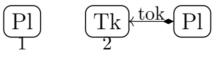
$$d = \nexists(\ \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\ ).$$

Then we have the following (see Examples 9 and 10).

$\mathrm{Shift}(y, d) = \nexists($  $) \wedge \ldots$

$\mathrm{Shift}(x, d) = \nexists($  $) \wedge \ldots$

$\mathrm{Left}(\varrho, \mathrm{Shift}(y, d)) = \nexists($  $) \wedge \ldots$

$\mathrm{Pres}(\varrho, d) = \nexists($  $) \wedge \ldots \Rightarrow \nexists($  $) \wedge \ldots$

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

**Lemma 6 (**Gua**).** For each rule $\varrho$ with interfaces $A$ and each condition $d$ over $A$, a condition $\mathrm{ac}_{gua} = \mathrm{Gua}(\varrho, d)$ can be constructed such that all transformations $g \Rightarrow_{\langle \varrho, \mathrm{ac}\rangle} h$ are *d-guaranteeing*, i.e., $h \models d$.

**Construction 4.** The construction Gua is as follows. For an rule $\varrho$ with interfaces $A$ and a condition $d$ over $A$, let $\mathrm{Gua}(\varrho, d) := \mathrm{Left}(\varrho, \mathrm{Shift}(y, d))$.

**Proof.** For every transformation $g \Rightarrow_{\langle \varrho, \mathrm{Gua}(\varrho, d)\rangle} h$, $(h \models d)$ iff $(h' \models \mathrm{Shift}(y, d))$ iff $(g' \models \mathrm{Left}(p, \mathrm{Shift}(y, d))$ iff $g' \models \mathrm{Gua}(\varrho, d)$. □

By Lemma 6, for every rule $\varrho$, we can construct the rule $\mathrm{gua}(\varrho, d) = \langle \varrho, \mathrm{Gua}(\varrho, d)\rangle$. The rule is *d-guaranteeing*, i.e., for all transformations $g \Rightarrow_{\mathrm{gua}(\varrho, d)} h$, $h \models d$.

**Bibliographic notes.** The basic transformations were first stated and proven for high-level transformations systems [HP09] and extensively used to prove correctness relative to conditions. The transformations are implemented in ENFORCE [AHPZ06] and Henshin [SBG$^+$17]. In Nassar et al. [NKAT20], a method to simplify constraint-preserving application conditions is presented. Their simplifications of the application conditions are based on three main concepts: (1) If the elements which are deleted (or added) by a rule are type-disjoint with the types of the constraint, i.e., they share no types, the application condition simplifies to `true`, (2) For increasing (or decreasing) rules, and positive (or negative) constraints, the application is `true`, (3) For negative constraints $\nexists C$ one may omit the cases where $C$ and the elements created by the rule overlap in at least one element. The simplified application conditions are proven to be logically equivalent to the original application condition. The results are proven to be correct for $\mathcal{M}$-adhesive categories and can be used to simplify the application conditions needed for our construction of condition-preserving application conditions (Lemma 5), as well.

# Chapter 3

# Graph generation

In meta-modelling, a general problem is how to generate instances of a meta-model. There are several approaches to instance generation: most of them are logic-oriented, some are rule-based (for an overview see, e.g., Radke et al. [RAB+15, RAB+18]). Our approach to instance generation is rule-based.

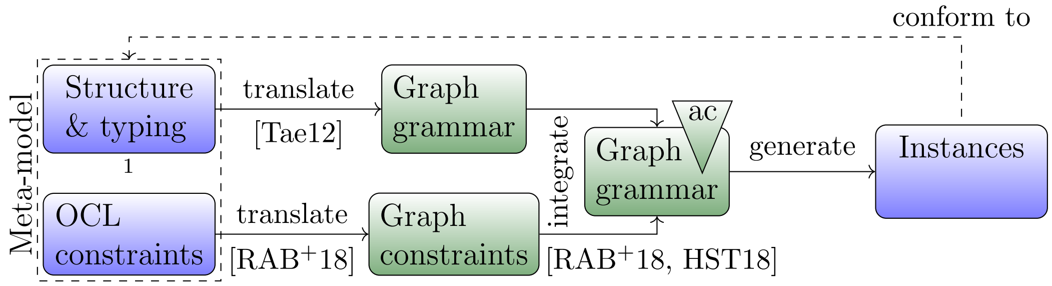


Figure 3.1: Instance generation of meta-models

(1) Translate the structure of the meta-model without OCL constraints and its typing into a (typed) graph grammar $GG$ (Taentzer [Tae12]) and OCL constraints into graph constraints $c$ (Radke et al. [RAB+18]).

(2) Integrate the graph constraint $c$ into the graph grammar $GG$ yielding a graph grammar $GG_c$ generating the graphs satisfying the graph constraints.

(3) Generate instances $I \in L(GG_c)$.

[1] The illustration shows the main components of a meta-model: the structure and the typing as well as OCL constraints. Of course, there are relations between the components. The OCL constraints are typed over the meta-model. The relations between the structure and typing of the meta-model and the OCL constraints are not drawn. For a definition of a meta-model, see Section 5.1.

In Radke et al. [RAB$^{+}$18], the integration of a graph constraint $c$ into a graph grammar $GG$ is done by replacing the rules $\varrho$ of the grammar by the corresponding $c$-guaranteeing rules. Unfortunately, this yields a grammar $GG_c$ generating a subset of all graphs satisfying the constraint $c$, i.e., $L(GG_c) \subsetneq L(GG) \cap [\![c]\!]$, and the inclusion is usually proper.

In this chapter, we look for a construction of a graph grammar $GG_c$ that generates exactly the set of all graphs of the original graph grammar that satisfy the constraint $c$: $L(GG_c) = L(GG) \cap [\![c]\!]$. We look for a grammar that "filters" exactly those graphs of the graph language that satisfy the constraint. In the following, we talk about the filter problem.

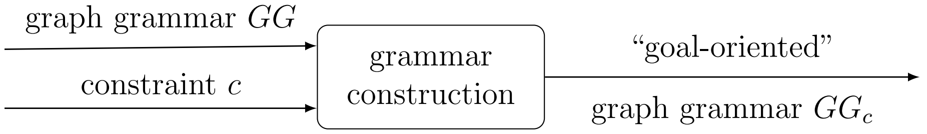
**Filter Problem**

**Given**: A graph grammar $GG$ and a graph constraint $c$.
**Question**: Does there exist a graph grammar $GG_c$ such that $L(GG_c) = L(GG) \cap [\![c]\!]$?

Moreover, we look for a "goal-oriented" (see Section 3.2) grammar that "filters" a subset of those graphs of the graph language that satisfy the constraint $c$: $L(GG_c) \subsetneq L(GG) \cap [\![c]\!]$. In the following, we talk about the weak filter problem.

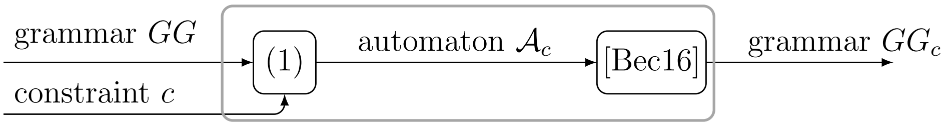
**Weak Filter Problem**

**Given**: A graph grammar $GG$ and a graph constraint $c$.
**Question**: Does there exist a "goal-oriented" graph grammar $GG_c$ such that $L(GG_c) \subseteq L(GG) \cap [\![c]\!]$?



We solve the (weak) filter problem for specific graph grammars $GG$ and specific graph constraints $c$. The construction is done in two steps (see below).

(1) Construct a "goal-oriented constraint automaton" $\mathcal{A}_c$ with $L(\mathcal{A}_c) \subseteq L(GG) \cap [\![c]\!]$.
(2) Construct a "goal-oriented" graph grammar $GG_c$ with $L(GG_c) = L(\mathcal{A}_c)$.



We illustrate our approach with Petri nets as the modeling language. We consider a (typed) graph grammar for generating Petri nets and graph constraints for restricting Petri nets. The example is a simplification of the one in Radke et al. [RAB$^+$18] for typed attributed graphs.

In Section 3.1, we sketch some basics on existential weakest liberal preconditions, containments, and minimizations. In Section 3.2, we introduce so-called constraint automata, present a backward construction, sketch some closure properties, consider the termination of the backward construction, and derive a goal-oriented graph grammar from the constraint automaton. In Section 3.3, we present some related concepts. In Section 3.4, we give a conclusion.

## 3.1 Existential weakest liberal preconditions

In this section, we sketch the prerequisites for our backward construction in Section 3.2: existential weakest liberal preconditions, similar to (universal) weakest liberal preconditions. Moreover, we introduce the syntactic operations containment and minimization which imply implication and equivalence, respectively.

The notion of (universal) weakest liberal preconditions is well-known (see, e.g., Apt and Olderog [AO91]).[2] In Habel and Pennemann [HP09], it is shown that a condition $c$ is a (universal) weakest liberal precondition of a rule relative to $d$ if, for all graphs $G$, $G \models c$ iff, for all transformations $G \Rightarrow H$ through the rule, $H \models d$.

We introduce existential weakest liberal preconditions and obtain a corresponding result. We use existential weakest liberal preconditions in the construction of a "constraint" automaton (Definition 15) and guarantee that, for each path in the automaton, there exists a derivation in the given graph grammar.

**Definition 10 (existential weakest liberal precondition).** Given a constraint $d$ and a rule $\varrho$, a condition $c$ is an *(existential) liberal precondition* of a rule $\varrho$ relative to a condition $d$, if, for all $G$ satisfying $c$, there exists some $G \Rightarrow_\varrho H$ such that $H \models d$ and an (existential) *weakest* liberal precondition of $\varrho$ relative to $d$, if any (existential) liberal precondition of $\varrho$ relative to $d$ implies $c$.

The following characterization points out a simple proof scheme for (existential) weakest liberal preconditions.

**Fact 3 (characterization).** A condition c is an existential weakest liberal precondition of $\varrho$ relative to $d$ if, for all $G$, $G \models c$ iff there exists some $G \Rightarrow_\varrho H$ such that $H \models d$.

[2] In [AO91], the term weakest liberal preconditions is used to ensure partial correctness, the term weakest preconditions for ensuring total correctness.

**Proof.** Analogously to (universal) weakest liberal preconditions [HP09]: Assume, for all graphs $G$, $G \models c$ iff there exists some $G \Rightarrow_\varrho H$ such that $H \models d$. By Definition 10, $c$ is an existential liberal precondition. It remains to show that for any other existential liberal precondition $c'$, $c'$ implies $c$. Let $G$ be an arbitrary graph and assume $G \models c'$. According to Definition 10, there is some $G \Rightarrow_\varrho H$ such that $H \models d$. Using the assumption, we have $G \models c$. For all graphs $G$, $G \models c'$ implies $G \models c$, i.e., $c' \implies c$. Thus, $c$ is a existential weakest liberal precondition. □

Similar to the weakest liberal preconditions of Habel and Pennemann [HP09], we construct an existential weakest liberal precondition.

**Lemma 7 (Wlp$_\exists$).** There is a construction $\mathrm{Wlp}_\exists$ such that, for every rule $\varrho$ and every constraint $d$, $\mathrm{Wlp}_\exists(\varrho, d)$ is an existential weakest liberal precondition of $\varrho$ relative to $d$.

**Proof.** Analogously to the corresponding proof for weakest liberal precondition in Habel and Pennemann [HP09]. □

The existential weakest liberal precondition of a rule and a constraint is constructed by the basic transformations from graph constraints to right application conditions (Shift), right to left application conditions (Left), and application conditions to constraints (C$_\exists$).

**Construction 5.** For a rule $\varrho = \langle p, \mathrm{ac}\rangle$, $\mathrm{Wlp}_\exists(\varrho, d) := \mathrm{C}_\exists(\mathrm{Left}(\varrho, \mathrm{Shift}(b, d) \wedge \mathrm{Appl}(\varrho))$ where $b\colon \emptyset \hookrightarrow \mathrm{Rhs}(\varrho)$, and $\mathrm{C}_\exists$ and Appl are defined as follows. For application conditions ac over $L$, $\mathrm{C}_\exists(\mathrm{ac}) := \exists\,(\emptyset \hookrightarrow L, \mathrm{ac})$. $\mathrm{Appl}(\varrho) = \mathrm{Dang}(p) \wedge \mathrm{ac}$ and $\mathrm{Dang}(p) = \wedge_{a \in A} \nexists\, a$ where $A$ ranges over all minimal morphisms $a\colon L \hookrightarrow L'$ such that $\langle K \hookrightarrow L, a\rangle$ has no pushout complement. The latter condition expresses the *dangling condition* (e.g., see Habel and Pennemann [HP09]).

**Example 12.** For the rule `AddTok` $= \boxed{\mathrm{Pl}} \Rightarrow \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}$ and the constraint $2\mathrm{tok} = \exists\,(\boxed{\mathrm{Tk}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}})$, the constraint $\mathrm{Wlp}_\exists(\texttt{AddTok}, 2\mathrm{tok})$ is constructed as follows: Shift the constraint 2tok over the morphism $b$ from the empty graph to the right-hand side of `AddTok`, shift the obtained right application condition over the rule 2tok to the left (Left), and transform the obtained left application condition to a constraint (C$_\exists$) where the match of the rule is marked in blue.

$$\begin{array}{lll}
(1)\quad \mathrm{Shift}(b, 2\mathrm{tok}) & = & \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}\quad \boxed{\mathrm{Tk}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}) \vee \ldots \\
 & & \vee\exists\,(\boxed{\mathrm{Tk}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}) = \mathrm{ac}_R \\
(2)\quad \mathrm{Left}(\texttt{AddTok}, \mathrm{ac}_R) & = & \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}) = \mathrm{ac}_L \\
(3)\quad \mathrm{C}_\exists(\mathrm{ac}_L) & = & \exists\,(\boxed{\mathrm{Pl}}, \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}) \equiv \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}})
\end{array}$$

The conditions mean that (1) there exists a place with a token (the comatch of the rule) and a place with two tokens or … or a place with two tokens, (2) there exists a place (the match of the rule) with one token, and (3) there exists a place with one token, respectively.

Containment of conditions is based on the existence of an injective morphism. The existence of an injective morphism between two conditions guarantees the implication of the conditions. Later on, we use containment to simplify the existential weakest liberal preconditions.

**Definition 11 (containment).** The *containment* of conditions $d_1$ and $d_2$ over $A$, denoted $d_1 \sqsubseteq d_2$, is defined as follows. For every condition $d_2$, $\texttt{true} \sqsubseteq d_2$. For $d_i = \exists\,(A \overset{a_i}{\hookrightarrow} C_i, c_i)$ $(i = 1, 2)$, $d_1 \sqsubseteq d_2$ if there is an injective morphism $b\colon C_1 \hookrightarrow C_2$ such that $b \circ a_1 = a_2$ and $\mathrm{Shift}(b, c_1) \sqsubseteq c_2$.

$$\begin{tikzcd}
 & C_1 \arrow[dd, hook, "b"] \arrow[r, phantom, "\blacktriangleleft"] & c_1 \\
A \arrow[ur, hook, "a_1"] \arrow[dr, hook, "a_2"'] \arrow[r, phantom, "="] & & \\
 & C_2 \arrow[r, phantom, "\blacktriangleleft"] & c_2
\end{tikzcd}$$

For negative conditions $d_i = \neg c_i$ $(i = 1, 2)$, $d_1 \sqsubseteq d_2$ if $c_2 \sqsubseteq c_1$. For conjunctive conditions $d_i = \wedge_{j \in I_i} c_{ij}$ $(i = 1, 2)$, $d_1 \sqsubseteq d_2$ if there is an injective function $f\colon I_1 \hookrightarrow I_2$ such that $c_{1j} \sqsubseteq c_{2f(j)}$ for all $j \in I_1$. A constraint $c_2$ *subsumes* $c_1$, $c_2 \sqsupseteq c_1$, if $c_1$ contains $c_2$. A constraint $c_1$ *properly* contains in $c_2$ ($c_2$ *properly* subsumes $c_1$), $c_1 \sqsubset c_2$ ($c_2 \sqsupset c_1$), if the injective morphism is not an isomorphism.

**Example 13.** Consider the constraint $d = d_1 \vee d_2$ with $d_1 = \exists\,(\ \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ )$ and $d_2 = \exists\,(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ )$ meaning there exists a Pl-node with two Tk-nodes and the connecting containment edges or there exists a Pl-node with one Tk-node and the connecting containment edge. There is an injective, non-isomorphic morphism from the graph of $d_2$ to the graph of $d_1$, i.e., $d_2 \sqsubset d_1$, and the condition $d_2$ contains $d_1$ ($d_1$ subsumes $d_2$).

**Lemma 8.** For conditions $d_1, d_2$, if $d_1 \sqsubseteq d_2$, then $d_2 \Rightarrow d_1$.

**Proof.** By induction over the size of graph conditions. Let $d_1 \sqsubseteq d_2$.
If $d_1 = \texttt{true}$, for every injective morphism $p\colon A \hookrightarrow G$, $p \models d_2$ implies that $p \models \texttt{true}$, i.e., $d_2 \Rightarrow \texttt{true}$. If $d_i = \exists\,(a_i, c_i)$ with $a_i\colon A \hookrightarrow C_i$ $(i = 1, 2)$, then there is an injective morphism $b\colon C_1 \hookrightarrow C_2$ such that $b \circ a_1 = a_2$ and $\mathrm{Shift}(b, c_1) \sqsubseteq c_2$. If $p\colon A \hookrightarrow G \models d_2$, then there is an injective morphism $q_2\colon C_2 \hookrightarrow G$ such that

$q_2 \circ a_2 = p$ and $q_2 \models c_2$. Define now $q_1 = q_2 \circ b$. Then $q_1$ is injective and $q_1 \circ a_1 = q_2 \circ b \circ a_1 = q_2 \circ a_2 = p$. By induction hypothesis, $c_2 \Rightarrow \mathrm{Shift}(b, c_1)$. Since $q_2 \models c$, $q_2 \models \mathrm{Shift}(d, c_1)$. By the Shift Lemma 3, $q_1 \models c_1$. By definition of $\models$, $p \models \exists\,(a_1, c_1) = d_1$. Thus, $d_2 \Rightarrow d_1$.

$$\begin{tikzcd}
 & & C_1 \arrow[dd, hook, "b"] \arrow[llld, hook, bend right, "q_1"'] \arrow[r, phantom, "\lhd"] & c_1 \\
G & A \arrow[l, hook, "p"'] \arrow[ur, hook, "a_1"] \arrow[dr, hook, "a_2"'] \arrow[ul, phantom, "="] \arrow[r, phantom, "="] \arrow[dl, phantom, "="] & & \\
 & & C_2 \arrow[llu, hook, bend left, "q_2"] \arrow[r, phantom, "\lhd"] & c_2
\end{tikzcd}$$

For negation and conjunction, the statement follows directly from the definitions and the inductive hypothesis. If $d_i = \neg c_i$ $(i = 1, 2)$, then $c_2 \sqsubseteq c_1$. By induction hypothesis, $c_1 \Rightarrow c_2$. Hence, $d_2 = \neg c_2 \Rightarrow \neg c_1 = d_1$. If $d_i = \wedge_{j \in I_i} c_{ij}$ $(i = 1, 2)$, then there is an injective function $f\colon I_1 \hookrightarrow I_2$ such that $c_{1j} \sqsubseteq c_{2f(j)}$ for all $j \in I_1$. By induction hypothesis, $c_{2f(j)} \Rightarrow c_{1j}$ for all $j \in I_1$. Then $\wedge_{j \in I_1} c_{2f(j)} \Rightarrow \wedge_{j \in I_1} c_{1j}$ and, by $f(I_1) \subseteq I_2$, $d_2 \Rightarrow d_1$. □

**Definition 12 (minimization).** Let $d = \bigvee_{i \in I} d_i$ be a finite disjunction of conditions. Then $\mathrm{Min}(d) = \bigvee_{i \in I'} d_i$ where $I' = \{i \in I \mid \nexists\, d_j .\, d_j \sqsubseteq d_i \text{ for all } j \in I\}$.

**Lemma 9.** For disjunctive conditions $d$, $\mathrm{Min}(d) \equiv d$.

**Example 14.** We continue with Example 13. Since $d_2$ contains $d_1$ ($d_1$ subsumes $d_2$), the disjunction can be minimized. This yields the constraint $d' = \bigvee d_2$. The constraint is equivalent to the original constraint $d$ from Example 13: each graph consisting of at least one Pl-node, a Tk-node and the connecting containment edge, satisfies $d_2$ and consequently $d$. Vice versa, each graph, which does not satisfy $d_2$, i.e., which does not consist of at least on Pl-node, a Tk-node and the connecting containment edge, does not satisfy $d_1$.

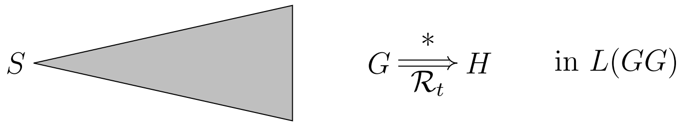
**Proof.** "$\Rightarrow$". Let $p \models \mathrm{Min}(d) = \vee_{i \in I'} d_i$. Then $p \models \vee_{i \in I} d_i$ since $I' \subseteq I$.

"$\Leftarrow$". Let $p \models \neg\mathrm{Min}(d) = \wedge_{i \in I'} \neg d_i$. By definition of the semantics, $p \models \neg d_i$ for all $i \in I'$. It remains to show $p \models \neg d_i$ for all $i \notin I'$. For $i \notin I'$, there is some index $j \in I'$ such that $d_j \sqsubseteq d_i$. By Lemma 8, we have $d_i \Rightarrow d_j \equiv \neg d_j \Rightarrow \neg d_i$. Now $p \models \neg d_j$ implies $p \models \neg d_i$. Thus, $p \models \neg d_i$ for all $i \in I$. Consequently, $\mathrm{Min}(d) \equiv d$. □

## 3.2 Filtering through constraints

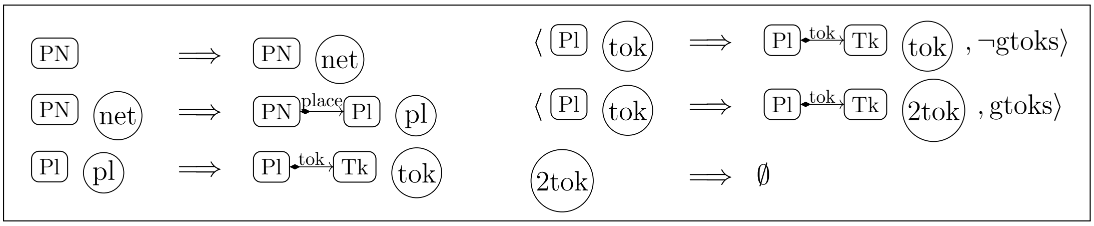
In this section, we investigate the filter problem for graph grammars and constraints. We introduce so-called constraint automata, present a backward construction for constraint automata, sketch some closure properties closely related to the ones in formal language theory, consider the termination of the backward construction, and derive a goal-oriented graph grammar from the constraint automaton. The section is concluded by a Filter Theorem, summarizing the result.

**Goal-oriented grammars.** Our aim is to construct a “goal-oriented” graph grammar without **dead end**, i.e., a grammar with a terminating rule set, a subset of the given one, such that, for each reachable graph, there exists a derivation to a graph satisfying a constraint using the terminating rule set (see Figure below) The advantage is that the application of the rules is backtracking-free.


$S$
$G \overset{*}{\underset{\mathcal{R}_t}{\Longrightarrow}} H$
in $L(GG)$

**Definition 13 (goal-oriented graph grammar).** Let $GG = (\mathcal{N}, \mathcal{R}, S)$ be a graph grammar. The grammar $GG$ is *goal-oriented* if there is a terminating[3] rule set $\mathcal{R}_t \subseteq \mathcal{R}$ such that, for all derivable graphs, there is a transformation to a terminal graph using rules form $\mathcal{R}_t$.

**Example 15.** Consider the graph grammar for Petri nets from Example 1 equipped with the nonterminal symbols net, pl, tok, 2tok and the following set of rules, where pl abbreviates place.



The grammar shall generate a graph consisting of a place with at least two tokens. The grammar is goal-oriented: starting from the start graph $\boxed{\text{PN}}$, the grammar adds a nonterminal net. There is a derivation via the grammar, which adds a place to the net, then a token to the place, afterwards a token, provided that the place already has a token. The latter is ensured by the application condition gtoks $= \exists\,(\,\boxed{\text{Pl}}\quad \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,) \vee \exists\,(\,\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,)$. Finally, the non-terminal 2tok is deleted, yielding a terminal graph.

[3] A rule set $\mathcal{R}$ is *terminating* if there is no infinite transformation $G_0 \underset{\mathcal{R}}{\Rightarrow} G_1 \underset{\mathcal{R}}{\Rightarrow} G_2 \ldots$

## Constraint automata

In the following, we introduce constraint automata based on finite automata having constraints as states. Given a graph grammar, a constraint automaton consists of a finite automaton (see Hopcroft and Ullman [HU79]), with constraints as states and “restricting” rules as input symbols. The graph language of the automaton is the set of all graphs derived from the start graph by applying the sequences of rules accepted by the finite automaton.

**Definition 14 (restricting rule).** Given a rule $\varrho = \langle p, \mathrm{ac}\rangle$ with application condition and an application condition $\mathrm{ac}'$, then the rule $\langle p, \mathrm{ac} \wedge \mathrm{ac}'\rangle$ is said to be *$\varrho$-restricting*. A rule $\varrho$ is *$\mathcal{R}$-restricting* if $\varrho'$ is $\varrho$-restricting for some $\varrho \in \mathcal{R}$.

**Example 16.** In Figure 3.2, the rules $\langle \texttt{AddTok}, \mathrm{gtoks}\rangle$ and $\langle \texttt{AddTok}, \neg\mathrm{gtoks}\rangle$ are `AddTok`-restricting.

**Definition 15 (constraint automaton).** The *(constraint) automaton* for a graph grammar $GG = (\mathcal{R}, S)$ is a tuple $\mathcal{A} = (A, S)$ where $A = (\mathcal{C}, \mathcal{R}', \rightarrow, C_0, F)$ is a (finite) automaton, $\mathcal{C}$ a set of states (constraints), $\mathcal{R}'$ a finite set of $\mathcal{R}$-restricting rules, $\rightarrow$ a transition relation, $C_0, F \subseteq \mathcal{C}$ sets of initial and final states, respectively, and $S$ a start graph. The automaton is *goal-oriented* if there is a terminating transition relation $\rightarrow_t \subseteq \rightarrow$ such that, for all initial states and all reachable states, a final state is reachable using transitions from $\rightarrow_t$. The *graph language* of $\mathcal{A}$ is $L(\mathcal{A}) = \{G \mid \exists S \Rightarrow_w G \text{ for some } w \in L(A)\}$ where $L(A)$ is the set of all strings accepted by the finite automaton $A$. An automaton $\mathcal{A}_1 = (A_1, S)$ is a *subautomaton* of an automaton $\mathcal{A}_2 = (A_2, S)$ with $A_i = (\mathcal{C}_i, \mathcal{R}'_i, \rightarrow_i, C_{0i}, F_i)$, written $\mathcal{A}_1 \subseteq \mathcal{A}_2$, iff $\mathcal{C}_1 \subseteq \mathcal{C}_2$, $\mathcal{R}'_1 \subseteq \mathcal{R}'_2$, $\rightarrow_1 \subseteq \rightarrow_2$, $F_1 \subseteq F_2$, and $C_{01} \subseteq C_{02}$.

**Convention.** When drawing a constraint automaton, we draw the states with the names written inside a circle, transitions are drawn by arrows and are labelled by the rules (with application conditions). The start state is indicated by the arrow and the label start. The final state is drawn by double lines around the circle.

**Example 17.** Figure 3.2 shows a constraint automaton. The automaton shall generate all graphs consisting of a place with at least two tokens. The states are the constraints net, place, tok, 2tok written inside a circle. The transitions are drawn by arrows and are labelled by the rules (with application conditions) of the grammar for Petri nets from Example 1. E.g., the labels `AddTok` and gtoks denote the rule `AddTok` with application condition gtoks $= \exists\,(\,\boxed{\text{Pl}}\quad \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,) \vee \exists\,(\,\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,)$[4], meaning that outside of the match of `AddTok`, there is a Pl-node

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

with two Tk-nodes or at the Pl-node in consideration, there is one Tk-node and the containment tok-edge. The start state is the state net. The final state is the state 2tok.

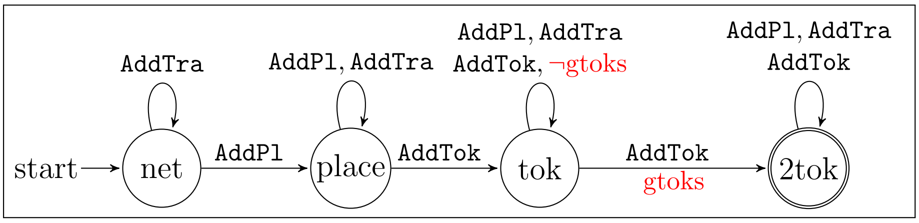


Figure 3.2: Constraint automaton

The automaton is goal-oriented: for all reachable states in the automaton there exists a terminating transition relation to the final state 2tok using this transition relation. Every path from the state net to state 2tok determines a transformation from a graph satisfying net to a graph satisfying 2tok.

## Backward construction

We construct a constraint automaton based on the construction of existential weakest liberal preconditions. Intuitively, the constraint automaton is constructed as follows (see Figure 3.3). Starting with a constraint $c$, for every rule $\varrho$, we construct the existential weakest liberal precondition $\mathrm{Wlp}_{\exists}(\varrho, c)$, bring it into normal form (which may be seen as a disjunction of constraints), minimize it, and, for each constraint $b$ in the disjunction, add $b$ to the constraint (and state) set and $b \to_{\varrho'} c$ with $\varrho' = \mathrm{gua}(\varrho, c)$ to the transition relation. We ignore constraints $b$ subsuming some constraint in $\mathcal{C}$. The constructed constraints become the states of the automaton, the constraint $c$ the final state, and the constraints which are satisfied for the start graph $S$ become the initial states.

**Assumption 2.** By Lemma 9, we may assume that the existential weakest liberal preconditions are in normal form and are minimized.

For a constraint set $\mathcal{C}$, $\mathrm{Max}(\mathcal{C})$ denotes the set of constraints $b$ in $\mathcal{C}$ that are maximal with respect to $\to$, i.e., there is no constraint $b' \in \mathcal{C}$ such that $b \to b'$.

**Construction 6 (backward construction with containment test).** Given a grammar $GG$ and a constraint $c$, we construct constraint automata as follows.

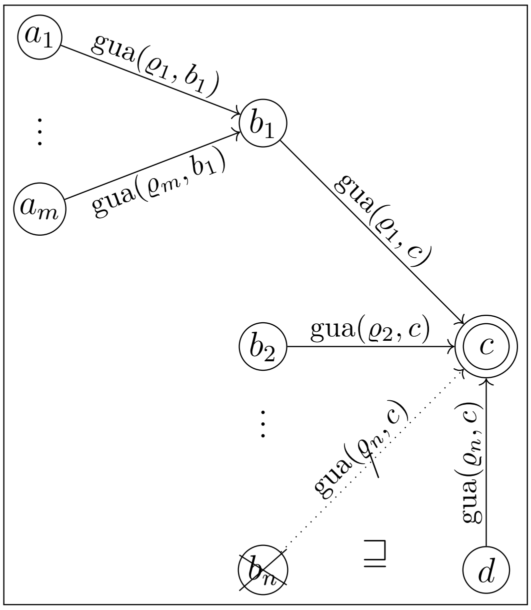


Figure 3.3: Overview of the idea for the backward construction

(1) Construct a sequence $\mathcal{C}_0, \mathcal{C}_1, \dots$ of constraint or state sets by $\mathcal{C}_0 = \{c\}$ and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{b \text{ in } \mathrm{Wlp}_\exists(\varrho, c') \mid \varrho \in \mathcal{R}, c' \in \mathrm{Max}(\mathcal{C}_i) \wedge \nexists\, b' \in \mathcal{C}_i.b \sqsupseteq b'\}$ where for a disjunction $c' = \vee_{i \in I} c_i$, $b$ in $c'$ abbreviates $b \in \{c_i \mid i \in I\}$. (For a rule and a maximal constraint of the set $\mathcal{C}_i$, we construct the existential weakest liberal precondition, bring it into normal form, minimize it, and add the components of the disjunction to the set $\mathcal{C}_{i+1}$ provided that they do not subsume another constraint of $\mathcal{C}_i$.)

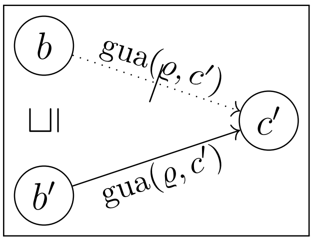
(2a) For $i \geq 0$, let $\mathcal{A}_{c,i} = (A_i, S)$ be the automaton with $A_i = (\mathcal{C}_i, \mathcal{R}', \rightarrow, C_0, \{c\})$, $\mathcal{R}' = \{\mathrm{gua}(\varrho, b) \mid \varrho \in \mathcal{R}, b \in \mathcal{C}_i\}$. For $\varrho \in \mathcal{R}$, $c' \in \mathcal{C}_i$, if $b$ in $\mathrm{Wlp}_\exists(\varrho, c')$ and $b \sqsupseteq b' \in \mathrm{Max}(\mathcal{C}_i)$, $b' \rightarrow_{\mathrm{gua}(\varrho,c')} c'$ is in $\rightarrow$. $C_0 = \mathrm{Max}(\{c_0 \in \mathcal{C} \mid S \models c_0\})$ and $\{c\}$ are the sets of initial and final states, respectively. (If the constraint $b$ is a component of the existential weakest liberal precondition of a rule $\varrho$ relative to the constraint $c'$ and $b$ subsumes a maximal constraint of $\mathcal{C}_i$, then there is a transition from $b'$ to $c'$ labelled with $\mathrm{gua}(\varrho, c')$.)

(2b) If $\mathcal{C}_i = \mathcal{C}_{i+1}$ for some $i \geq 0$, let $\mathcal{C} = \mathcal{C}_i$ and $\mathcal{A}_c = \mathcal{A}_{c,i}$.

**Remark.** 1. A constraint $b \in \mathcal{C}$ represents the class $[\![b]\!]$ of all graphs satisfying $b$.

2. By the $\mathrm{Wlp}_\exists(\varrho, c)$-construction, we obtain the existence of a direct transformation $G \Rightarrow_\varrho H$ such that $H \models c$. By the $\mathrm{gua}(\varrho, c)$-construction, we filter all those direct transformations that guarantee $c$: $G \Rightarrow_{\mathrm{gua}(\varrho,c)} H$ such that $H \models c$. In this way, for all direct transformations $G \Rightarrow_{\mathrm{gua}(\varrho,c)} H \wedge H \models c$.

3. An essential step for restricting the state exploration is the following. Whenever a constraint $c'$ is in the constraint set and $b \in \mathrm{Wlp}_\exists(\varrho, c')$ is a constraint such that $b$ subsumes a maximal constraint $b'$ in the constraint set, then the constraint $b$ and an edge from $b$ to $c'$ with label $\mathrm{gua}(\varrho, c')$ is not inserted (denoted by the dotted arrow below); instead an edge from $b'$ to $c'$ with label $\mathrm{gua}(\varrho, c')$ is inserted.



4. The set of initial states may be empty; in this case, $L(\mathcal{A}_{c,0})$, $L(\mathcal{A}_c)$ are empty.

5. For $i \geq 0$, $\mathcal{A}_{c,i} \subseteq \mathcal{A}_{c,i+1}$ and $L(\mathcal{A}_{c,i}) \subseteq L(\mathcal{A}_{c,i+1})$.

6. In general, the sequence $\mathcal{C}_0, \mathcal{C}_1, \ldots$ need not become stationary, hence the procedure need not terminate, but we can show that it terminates for our specific cases.

**Example 18.** Consider the grammar with the rules `AddPl`, `AddTra`, and `AddTok` in Example 1 and the positive constraint $2\mathrm{tok} = \exists\,(\boxed{\mathrm{Tk}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}})$ meaning that there exists a Pl-node and two Tk-nodes together with connecting containment tok-edges. Then the backward construction with containment test yields a finite automaton $\mathcal{A}_{2\mathrm{tok}}$ given in Figure 3.4.

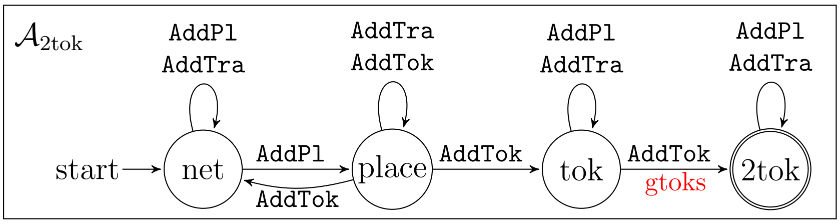


Figure 3.4: The automaton $\mathcal{A}_{2\mathrm{tok}}$, simplified.

The automaton $\mathcal{A}_{2\mathrm{tok}}$ is constructed as follows. Start with the constraint 2tok, construct, for every rule $p$ in the grammar and every constraint $b'$, the weakest liberal precondition $\mathrm{Wlp}_\exists(p, b')$, and subsume constraints. Continue with the new constraints in the constraint set. The construction terminates. The components of the weakest liberal preconditions in Figure 3.7 build the state set $\mathcal{C} = \{\mathrm{net}, \mathrm{place}, \mathrm{tok}, 2\mathrm{tok}\}$ where $\mathrm{net} = \exists\,(\boxed{\mathrm{PN}})$, $\mathrm{place} = \exists\,(\boxed{\mathrm{Pl}})$, and $\mathrm{tok} = \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}})$. The constraint $\mathrm{Wlp}_\exists(\texttt{AddTok}, \mathrm{net}) = \exists\,(\boxed{\mathrm{PN}}\ \ \boxed{\mathrm{Pl}})$ is ignored because it is subsumed by place.

The construction of the guaranteeing application condition Gua($\texttt{AddTok}, 2\text{tok}$) yields gtoks $= \exists\,(\,\boxed{\text{Pl}}\quad \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,) \vee \exists\,(\,\boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,)$[5], meaning that outside of the match of `AddTok`, there is a Pl-node with two Tk-nodes or at the Pl-node in consideration, there is one Tk-node and the containment tok-edge. All other application conditions are given in Figure 3.6. The initial state is net because $\boxed{\text{PN}} \models$ net and the final state is 2tok. The constraint automaton $\mathcal{A}_{2\text{tok}}$ is shown in Figure 3.5.

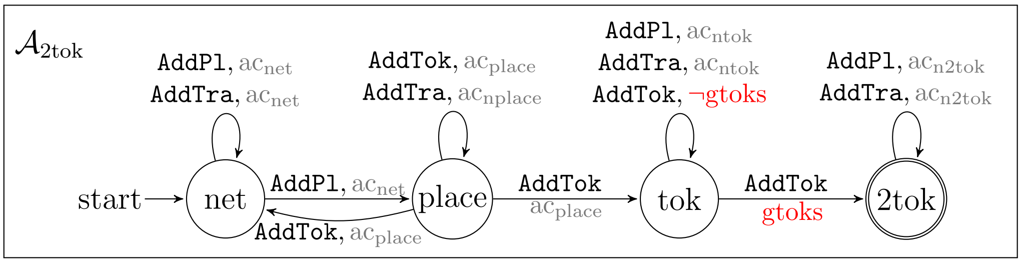

$\mathcal{A}_{2\text{tok}}$
AddPl, $\text{ac}_{\text{net}}$
AddTra, $\text{ac}_{\text{net}}$
AddTok, $\text{ac}_{\text{place}}$
AddTra, $\text{ac}_{\text{nplace}}$
AddPl, $\text{ac}_{\text{ntok}}$
AddTra, $\text{ac}_{\text{ntok}}$
AddTok, ¬gtoks
AddPl, $\text{ac}_{\text{n2tok}}$
AddTra, $\text{ac}_{\text{n2tok}}$
start
net
AddPl, $\text{ac}_{\text{net}}$
AddTok, $\text{ac}_{\text{place}}$
place
AddTok
$\text{ac}_{\text{place}}$
tok
AddTok
gtoks
2tok

Figure 3.5: The constraint automaton $\mathcal{A}_{2\text{tok}}$.

By the following simplification rules, the automaton may be simplified: Minimize the application condition Gua($p, b'$) according to containment (see Lemma 8). If Gua($p, b'$) $\equiv$ `true`, write $p$ instead of $\langle p, \text{Gua}(p, b')\rangle$. If Gua($p, b'$) is a positive application condition, then it can be ignored. By these rules one may obtain the constraint automaton presented in Figure 3.4.

From the structure, the constraint automata in Figure 3.5 and 3.4 look very similar. The one in Figure 3.4 is obtained from the one in Figure 3.5 by simplification: all gray application conditions are omitted, the red application conditions remain. They are essential: Whenever the application ¬gtoks is satisfied, the application of the rule `AddTok` does not create a second token on a place. Whenever the application condition gtoks is satisfied, the application of the rule `AddTok` creates a second token at a place.

**Lemma 10.** For arbitrary graph grammars $GG$ and arbitrary constraints $c$, the constraint automata $\mathcal{A}_{c,i}$ and $\mathcal{A}_c$ are goal-oriented and

1. $L(\mathcal{A}_{c,i}) \subseteq L(GG) \cap [\![c]\!]$ $(i \geq 0)$ and
2. $L(\mathcal{A}_c) = L(GG) \cap [\![c]\!]$ in case of termination.

The proof is based on the following lemma which relates paths in the constraint automaton $\mathcal{A}_{c,i}$ and transformations in the graph grammar $GG$.

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

$$
\begin{array}{llll}
\mathrm{Gua}(\mathtt{AddPl},\mathrm{2tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & = \mathrm{ac}_{\mathrm{n2tok}} \\
\mathrm{Gua}(\mathtt{AddTra},\mathrm{2tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & = \mathrm{ac}_{\mathrm{n2tok}} \\
\mathrm{Gua}(\mathtt{AddTok},\mathrm{2tok}) & = & \exists\,(\,\boxed{\mathrm{Pl}}\quad \boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & = \mathrm{ac}_{\mathrm{p2tok}} \\
 & & \vee\exists\,(\,\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & \vee\mathrm{ac}_{\mathrm{tok}} \\
\hline
\mathrm{Gua}(\mathtt{AddPl},\mathrm{tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & = \mathrm{ac}_{\mathrm{ntok}} \\
\mathrm{Gua}(\mathtt{AddTra},\mathrm{tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & = \mathrm{ac}_{\mathrm{ntok}} \\
\mathrm{Gua}(\mathtt{AddTok},\mathrm{tok}) & = & \exists\,(\,\boxed{\mathrm{Pl}}\quad \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) \vee \exists\,(\,\boxed{\mathrm{Pl}}\,) \equiv \exists\,(\,\boxed{\mathrm{Pl}}\,) & = \mathrm{ac}_{\mathrm{place}} \\
\hline
\mathrm{Gua}(\mathtt{AddPl},\mathrm{place}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\,) \vee \exists\,(\,\boxed{\mathrm{PN}}\,) \equiv \exists\,(\,\boxed{\mathrm{PN}}\,) & = \mathrm{ac}_{\mathrm{net}} \\
\mathrm{Gua}(\mathtt{AddTra},\mathrm{place}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\,) & = \mathrm{ac}_{\mathrm{nplace}} \\
\mathrm{Gua}(\mathtt{AddTok},\mathrm{place}) & = & \exists\,(\,\boxed{\mathrm{Pl}}\quad \boxed{\mathrm{Pl}}\,) \vee \exists\,(\,\boxed{\mathrm{Pl}}\,) \equiv \exists\,(\,\boxed{\mathrm{Pl}}\,) & = \mathrm{ac}_{\mathrm{place}} \\
\hline
\mathrm{Gua}(\mathtt{AddTra},\mathrm{net}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{PN}}\,) \vee \exists\,(\,\boxed{\mathrm{PN}}\,) \equiv \exists\,(\,\boxed{\mathrm{PN}}\,) & = \mathrm{ac}_{\mathrm{net}} \\
\mathrm{Gua}(\mathtt{AddPl},\mathrm{net}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{PN}}\,) \vee \exists\,(\,\boxed{\mathrm{PN}}\,) \equiv \exists\,(\,\boxed{\mathrm{PN}}\,) & = \mathrm{ac}_{\mathrm{net}} \\
\mathrm{Gua}(\mathtt{AddTok},\mathrm{net}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\,) & = \mathrm{ac}_{\mathrm{placen}}
\end{array}
$$

Figure 3.6: Guaranteeing application conditions.

$$
\begin{array}{llll}
\mathrm{Wlp}_{\exists}(\mathtt{AddPl},\mathrm{2tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & \sqsupseteq \mathrm{2tok} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddTra},\mathrm{2tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & \sqsupseteq \mathrm{2tok} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddTok},\mathrm{2tok}) & = & \exists\,(\,\boxed{\mathrm{Pl}}\quad \boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & \\
 & & \vee\exists\,(\,\boxed{\mathrm{Tk}}\xleftarrow{\mathrm{tok}}\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) \vee \exists\,(\,\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & \equiv \mathrm{tok} \\
\hline
\mathrm{Wlp}_{\exists}(\mathtt{AddPl},\mathrm{tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & \sqsupseteq \mathrm{tok} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddTra},\mathrm{tok}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,) & \sqsupseteq \mathrm{tok} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddTok},\mathrm{tok}) & = & \exists\,(\,\boxed{\mathrm{Pl}}\quad \boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,)\vee\exists\,(\,\boxed{\mathrm{Pl}}\xrightarrow{\mathrm{tok}}\boxed{\mathrm{Tk}}\,)\vee\exists\,(\,\boxed{\mathrm{Pl}}\,) & \equiv \mathrm{place} \\
\hline
\mathrm{Wlp}_{\exists}(\mathtt{AddPl},\mathrm{place}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\,) \vee \exists\,(\,\boxed{\mathrm{PN}}\,) & \equiv \mathrm{net} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddTra},\mathrm{place}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\,) & \sqsupseteq \mathrm{place} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddTok},\mathrm{place}) & = & \exists\,(\,\boxed{\mathrm{Pl}}\quad \boxed{\mathrm{Pl}}\,) \vee \exists\,(\,\boxed{\mathrm{Pl}}\,) & \equiv \mathrm{place} \\
\hline
\mathrm{Wlp}_{\exists}(\mathtt{AddTra},\mathrm{net}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{PN}}\,) \vee \exists\,(\,\boxed{\mathrm{PN}}\,) & \equiv \mathrm{net} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddPl},\mathrm{net}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{PN}}\,) \vee \exists\,(\,\boxed{\mathrm{PN}}\,) & \equiv \mathrm{net} \\
\mathrm{Wlp}_{\exists}(\mathtt{AddTok},\mathrm{net}) & = & \exists\,(\,\boxed{\mathrm{PN}}\quad \boxed{\mathrm{Pl}}\,) & \sqsupseteq \mathrm{place}
\end{array}
$$

Figure 3.7: Existential weakest liberal preconditions.

**Lemma 11 (correctness & completeness).** Let $GG$ be a graph grammar, $c$ a graph constraint, and $\mathcal{A}_{c,i}$ the constructed automaton. Let $w' \in \mathcal{R}'^*$ and $w \in \mathcal{R}^*$ be the sequences of restricted and underlying rules, respectively.

1. For all paths $g \rightarrow_{w'} h$ in $\mathcal{A}_{c,i}$ $(i \geq 0)$ and all graphs $G \models g$, there is a graph $H$ and a transformation $G \Rightarrow_w H$ in $GG$ such that $H \models h$.
2. In the terminating case, for all transformations $G \Rightarrow_w H$ in $GG$ such that $H \models h \in \mathrm{Max}(\mathcal{C})$, there is a path $g \rightarrow_{w'} h$ in $\mathcal{A}_c$ such that $G \models g \in \mathrm{Max}(\mathcal{C})$.

**Proof (By induction on the length of the path/transformation).**

1. By induction on the structure of $w'$. Let $f \rightarrow_{w'} h$ and $F \models f$.
**Induction basis.** For $w' = \varepsilon$, let $h = f$. Then $F \Rightarrow^0 H \models h$.
**Induction step.** For $w' = v'\varrho'$, $f \rightarrow_{v'\varrho'} h$ can be decomposed into $f \rightarrow_{v'} g \rightarrow_{\varrho'} h$ where $\varrho' = \mathrm{gua}(\varrho, h)$. By induction hypothesis, there is a transformation $F \Rightarrow_v G$ such that $G \models g$. By $g \in \mathrm{Wlp}_\exists(\varrho, h)$, we have $G \models \mathrm{Wlp}_\exists(\varrho, h)$. By Fact 3, there is a direct transformation $G \Rightarrow_\varrho H$ such that $H \models h$. Composing the transformations, we obtain a transformation $F \Rightarrow_{v\varrho} H$ such that $H \models h$.

2. By induction on the structure of $w$. Let Construction 6 be terminating and $\mathcal{A}_c$ the resulting constraint automaton. Let $F \Rightarrow_w H$ and $H \models h \in \mathrm{Max}(\mathcal{C})$.
**Induction basis.** For $w = \varepsilon$, $F \cong H$, $F \Rightarrow_\varepsilon H$ and $F \models f = h \in \mathrm{Max}(\mathcal{C})$.
**Induction step.** For $w = v\varrho$, the transformation $F \Rightarrow_{v\varrho} H$ can be decomposed into $F \Rightarrow_v G \Rightarrow_\varrho H$. By Fact 3 and Lemma 9, $G \Rightarrow_\varrho H.H \models h$ implies $G \models \mathrm{Wlp}_\exists(\varrho, h)$. Consequently, there is some $g' \in \mathrm{Wlp}_\exists(\varrho, h).G \models g'$. By Construction 6, there is some $g \in \mathrm{Max}(\mathcal{C}).g' \sqsupseteq g$ and, by Lemma 8, $g' \Rrightarrow g$. By the definition of $\Rrightarrow$, $G \models g'$ implies $G \models g$. By Construction 6 and $g \in \mathrm{Max}(\mathcal{C})$, there is a transition $g \rightarrow_{\varrho'} h$ in $\mathcal{A}_c$. By $F \Rightarrow_v G.G \models g \in \mathrm{Max}(\mathcal{C})$, the induction hypothesis can be applied yielding a path $f \rightarrow_{v'} g$ in $\mathcal{A}_c$ such that $F \models f \in \mathrm{Max}(\mathcal{C})$. Composing the paths, we obtain a path $f \rightarrow_{w'} h$ such that $F \models f \in \mathrm{Max}(\mathcal{C})$.

$$\begin{tikzcd}
f \arrow[r, "v'"'] & g \arrow[r, "\varrho'"'] & h \\
\perp\!\!\perp \quad \mathrm{IH} & \perp\!\!\perp \ \mathrm{Wlp}_\exists & \perp\!\!\perp \\
F \arrow[r, Rightarrow, dashed, "v"'] & G \arrow[r, Rightarrow, dashed, "\varrho"'] & H
\end{tikzcd}
\qquad
\begin{tikzcd}
F \arrow[r, Rightarrow, "v"'] & G \arrow[r, Rightarrow, "\varrho"'] & H \\
\top\!\!\top \quad \mathrm{IH} & \top\!\!\top \ \mathrm{Wlp}_\exists & \top\!\!\top \\
f \arrow[r, dashed, "v'"'] & g \arrow[r, dashed, "\varrho'"'] & h
\end{tikzcd}$$

□

**Proof (of Lemma 10).** The statements follow immediately from Lemma 11:

1. If $G \in L(\mathcal{A}_{c,i})$, then $S \Rightarrow_w G$ for some $w' \in L(A_i)$. Then there is a path $c_0 \rightarrow_{w'} c$ from $c_0 \in C_{0,i}$ to $c$. Since $w' = \varepsilon$ ($S \cong G$ and $c_0 = c$) or the last rule in $w$ is $c$-guaranteeing, we have $G \models c$. Thus, $G \in L(GG) \cap [\![c]\!]$.

2. In case Construction 6 terminates, the first inclusion follows from the first statement. The second inclusion is as follows. If $G \in L(GG) \cap [\![c]\!]$, then there is a transformation $S \Rightarrow_w G$ in $GG$ such that $G \models c$. By Lemma 11, there is a path $c_0 \rightarrow_{w'} c$ in $\mathcal{A}_c$ such that $S \models c_0$. By construction, $c_0 \in C_0$. Thus, $w \in L(A)$ and $G \in L(\mathcal{A}_c)$.

3. The constraint automata are goal-oriented: By the backward construction, the automata are connected and all directed paths end in the final state. Let now $\mathcal{T}$ be a spanning tree, i.e., a subautomaton which is both a tree and which contains all the states of the automaton. Then the transition relation $\rightarrow_t := \rightarrow_{\mathcal{T}}$ is terminating and, for all reachable states, there is a path to the final state. □

## Termination

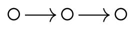
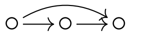
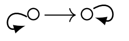
The question remains, under which assumptions the backward construction terminates. Unfortunately, in general, for non-deleting graph grammars and positive constraints, the construction does not terminate. The reason for this, is that the construction of the existential weakest liberal preconditions may yield an infinite sequence of new states (see Example 19). Adding the requirement "$n$-bounded path", a slightly modified backward construction terminates.

**Definition 16 (increasing, $n$-bounded graph grammar).** A graph grammar is *increasing* if all underlying plain rules are increasing, i.e., for $\langle L \hookleftarrow K \hookrightarrow R\rangle$, $L \cong K \subset R$. A graph grammar is *$n$-bounded path* if all generated graphs are $n$-bounded path, i.e., all paths have length less than or equal to $n$. A graph constraint of the form $\exists\, C$ is *$n$-bounded path*, if $C$ is $n$-bounded path.

**Lemma 12 (termination).**

1. For increasing graph grammars and positive constraints, the backward construction, in general, does not terminate.
2. If, additionally, the graph grammar is $n$-bounded path, there exists a terminating, slightly modified backward construction.

**Construction 7 (backward construction with bounded-path test).** Modify Construction 6 by replacing step (1) by step (1'): Construct a sequence $\mathcal{C}_0, \mathcal{C}_1, \ldots$ of constraint or state sets by $\mathcal{C}_0 = \{c\}$ and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{b \in \mathrm{Wlp}_\exists(\varrho, c') \mid \varrho \in \mathcal{R},\, c' \in \mathcal{C}_i \wedge \nexists b' \in \mathcal{C}_i . b \sqsupseteq b', b \text{ is } n\text{-bounded path}\}$.

**Example 19.** For the 1-bounded path grammar with the rule `AddEdge` = $\langle$  $\Rightarrow$  $\rangle$ and the constraint $c = \exists$  , Construction 6 does not terminate:

$$\begin{aligned}
\mathrm{Wlp}_\exists(\texttt{AddEdge}, c) &= \exists\ \circlearrowleft\!\circ\!\to\!\circ\!\to\!\circ\!\circlearrowright & &=: c_1\\
\mathrm{Wlp}_\exists(\texttt{AddEdge}, c_1) &= \exists\ \circlearrowleft\!\circ\!\to\!\circ\!\to\!\circ\!\to\!\circ\!\circlearrowright \ \vee c_1 & &=: c_2\\
\mathrm{Wlp}_\exists(\texttt{AddEdge}, c_2) &= \exists\ \circlearrowleft\!\circ\!\to\!\circ\!\to\!\circ\!\to\!\circ\!\to\!\circ\!\circlearrowright \ \vee c_2 & &=: c_3\\
\vdots \qquad & \quad \vdots
\end{aligned}$$

Starting with the constraint $c$, it produces an infinite sequence

$$\ldots \to \exists\ \circlearrowleft\!\circ\!\to\!\circ\!\to\!\circ\!\to\!\circ\!\to\!\circ\!\circlearrowright \ \to \exists\ \circlearrowleft\!\circ\!\to\!\circ\!\to\!\circ\!\to\!\circ\!\circlearrowright \ \to \exists\ \circlearrowleft\!\circ\!\to\!\circ\!\to\!\circ\!\circlearrowright \ \to \exists\ \circlearrowleft\!\circ\!\to\!\circ\!\circlearrowright$$

of positive constraints $\exists\, C_n$ requiring the existence of a path of length $n$ with two loops at the start and end node. Since there does not exist an injective morphism from $C_n$ to $C_{n+1}$, the constraint $\exists\, C_{n+1}$ is not contained in $\exists\, C_n$. In contrast, Construction 7 terminates: by the containment operator $\sqsupseteq$, each state $b'$, whose length is not $n$-bounded, is subsumed, and therefore not added to the state set.

The proof of Lemma 12 makes use of the notions of well-quasi-ordering and monotony. A binary relation $\preceq$ defined on a set $Q$ is a *quasi-ordering* (Ding [Din92]) if it is reflexive and transitive. A sequence $q_1, q_2, \ldots$ of members of $Q$ is called a *good sequence* (with respect to $\preceq$) if there exist $i < j$ such that $q_i \preceq q_j$. It is a *bad sequence* if otherwise. We call $(Q, \preceq)$ a *well-quasi-ordering* (or a *wqo*) if there is no infinite bad sequence. Let $T$ be a transition system with a preorder $\preceq$ defined on its states. $T$ is *monotone* w.r.t. to $\preceq$ if, for any states $c_1, c_2$ and $c_3$, with $c_1 \preceq c_2$ and $c_1 \to c_3$, there exists a state $c_4$ such that $c_3 \preceq c_4$ and $c_2 \to c_4$.

**Proof (of Lemma 12).** 1. The statement follows immediately from the undecidability of the coverability problem for non-deleting graph grammars (Bertrand et al. [BDK$^+$12][6]). The *Coverability Problem* is as follows. Given a graph grammar $GG = (\mathcal{R}, S)$ and a graph $C$, is there a graph $H$ such that $S \Rightarrow^*_{\mathcal{R}} H$ and $C \sqsubseteq H$, i.e., there is an injective morphism from $C$ to $H$. Assume the automaton construction terminates for increasing graph grammars $GG$ and positive constraints $c = \exists\, C$ with automaton $\mathcal{A}_c$. Then, the coverability problem for $GG$ and final graph $C$ is decidable: If in $\mathcal{A}_c$, there is a path from an initial state to the final state $c$, then the output is yes, and no, otherwise. Contradiction [BDK$^+$12, Proposition 13].

2. Correctness and completeness of Construction 7 follows from the correctness and completeness of Construction 6. Termination follows from the well-known

[6] In Bertrand et al. [BDK$^+$12], a slight extension of the single-pushout approach is considered, but the simulation of a deterministic Turing machine is done by a non-deleting double-pushout graph transformation system, see, e.g., [EHK$^+$97].

fact, that the subgraph relation for $n$-bounded path graphs is a well-quasi-order for $n$-bounded path graphs (Ding [Din92]) and the constraint system is monotone with respect to the subgraph relation (Abdulla and Jonsson [AJ01]). We use the subgraph relation $\sqsubseteq$ on graphs with $C_1 \sqsubseteq C_2$ if there exists an injective morphism from $C_1$ to $C_2$. This corresponds directly to the containment relation on positive constraints. Since the subgraph relation is a well-quasi-order for $n$-bounded path graphs, the containment relation is also a well-quasi-order.
Let $G, H$ be graphs, $p$ be a rule, and $G$ be a subgraph of $H$. Let $p$ be applicable on $G$, then there exists an injective morphism $g : L \hookrightarrow G$ from the left-hand side of $p$ to $G$. Since $G$ is a subgraph of $H$ there exists an injective morphism $m : G \hookrightarrow H$. Composing $g$ and $m$, we obtain the morphism $h$ from $L$ to $H$. The application of $p$ replaces $L$ by $R$ in both graphs $G$ and $H$ resulting in $G'$ and $H'$, respectively. Since $p$ is increasing, it follows that $G'$ is a subgraph of $H'$, thus the constraint system is monotone w.r.t. the subgraph relation. Suppose Construction 7 would not terminate for $n$-bounded path grammars. Then there would exist an infinite path $\cdots \rightarrow\rightarrow c_2 \rightarrow c_1 \rightarrow c_0$ where each $c_i$ is $n$-bounded path $(i \geq 0)$. By Construction 7, each $c_i$ is $n$-bounded path. Moreover, $\nexists i < j.\, c_i \sqsubseteq c_j$, because otherwise $c_j$ would immediately be subsumed. Since the $\sqsubseteq$ is a well-quasi-order, there must exist $i < j.\, c_i \sqsubseteq c_j$. This is a contradiction. Thus, the construction terminates. □

## Closure properties

Similar to formal language theory, we define deterministic constraint automata and transform nondeterministic constraint automata into deterministic ones. Moreover, it is shown that constraint automata are closed under the Boolean operations complementation and product construction.

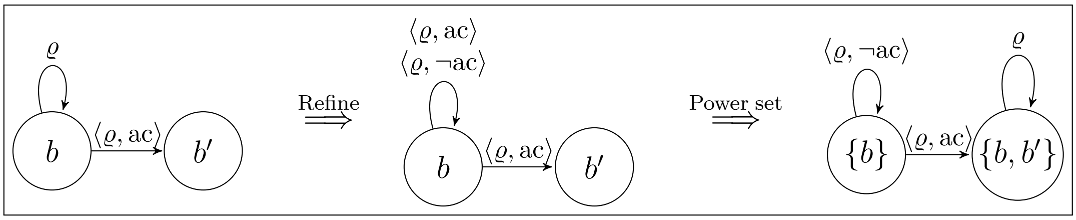
**Definition 17 (deterministic automata).** A constraint automaton $\mathcal{A} = (A, S)$ is *deterministic* if for each constraint $b \in \mathcal{C}$ and each rule $\varrho$ in the automaton, the application conditions in $b \rightarrow_{\langle \varrho, \mathrm{ac}_i \rangle} b_i$ are disjoint. Two application conditions ac and ac$'$ are *disjoint* if the sets $[\![\mathrm{ac}]\!]$ and $[\![\mathrm{ac}']\!]$ are disjoint.

**Example 20.** Examples of deterministic automata are given in Figures 3.9 and 3.10.

**Lemma 13.** For every constraint automaton $\mathcal{A}$, a deterministic automaton $\mathcal{A}'$ can be effectively constructed such that $L(\mathcal{A}) = L(\mathcal{A}')$.

**Construction 8.** By a refined power-set construction (see, e.g., Hopcroft and Ullman [HU79]). Let AC be the set of application conditions occurring in the automaton and $\mathrm{AC}'$ the refined set such that all application conditions are disjoint. Refine each transition $b \to_{\langle \varrho, \mathrm{ac} \rangle} b'$ by the transitions $b \to_{\langle \varrho, \mathrm{ac}_i \rangle} b'$ with $\mathrm{ac} = \bigwedge \mathrm{ac}_i$, $\mathrm{ac}_i \in \mathrm{AC}'$ and apply the power-set construction.

**Remark.** In most of our examples, we have an unrestricted rule and a rule restricted by an application condition ac. By the refined power-set construction, we get the simplification illustrated in Figure 3.8.



$\varrho$
$b$
$\langle \varrho, \mathrm{ac} \rangle$
$b'$
Refine
$\langle \varrho, \mathrm{ac} \rangle$
$\langle \varrho, \neg\mathrm{ac} \rangle$
$b$
$\langle \varrho, \mathrm{ac} \rangle$
$b'$
Power set
$\langle \varrho, \neg\mathrm{ac} \rangle$
$\varrho$
$\{b\}$
$\langle \varrho, \mathrm{ac} \rangle$
$\{b, b'\}$

Figure 3.8: Illustration of the refined power-set construction

Different transitions between the same constraints are drawn by one line.

**Example 21 (power-set construction).** By the power-set construction, the automaton $\mathcal{A}_{\mathrm{2tok}}$ in Example 18 can be transformed into an equivalent deterministic automaton $\mathcal{A}'_{\mathrm{2tok}}$ given in Figure 3.9.

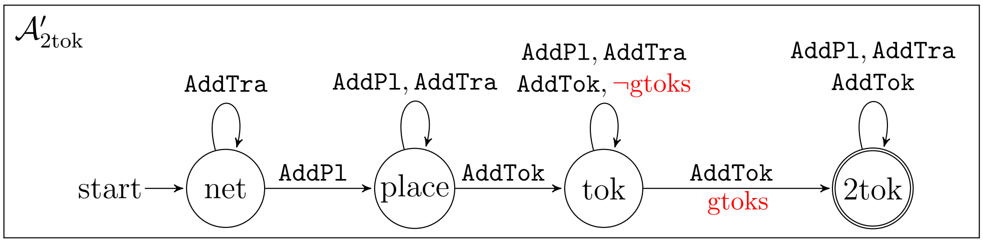


Figure 3.9: The deterministic automaton $\mathcal{A}'_{\mathrm{2tok}}$.

The automation $\mathcal{A}'_{\mathrm{2tok}}$ is deterministic: the restricted rules $\langle \texttt{AddTok}, \mathrm{gtoks} \rangle$ and $\langle \texttt{AddTok}, \neg\mathrm{gtoks} \rangle$ are distinct because the application conditions gtoks and ¬gtoks are distinct. The sets $\{\mathrm{net}\}$, $\{\mathrm{net}, \mathrm{place}\}$, $\{\mathrm{net}, \mathrm{place}, \mathrm{tok}\}$, $\{\mathrm{net}, \mathrm{place}, \mathrm{tok}, \mathrm{2tok}\}$ are represented by their maximum, i.e., net, place, tok and 2tok, respectively. Note that we draw only states accessible from the initial state. Moreover, we do not draw the transitions to the empty state.

**Proof.** The refinement step and the power-set construction do not change the semantics of the constraint automaton. By the power-set construction, the automaton is deterministic. □

**Example 22 (deterministic automaton).** The deterministic automaton $\mathcal{A}'_{\text{tok}}$ may be obtained from automaton $\mathcal{A}'_{2\text{tok}}$ of Example 21 by deleting the state 2tok and all incident edges (see Figure 3.10).

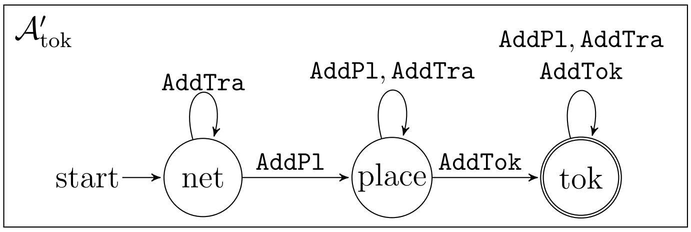


Figure 3.10: The deterministic automaton $\mathcal{A}'_{\text{tok}}$.

**Lemma 14.** The languages of constraint automata are closed under complement, intersection, and union.

**Idea.** The construction makes use of the complement and a slightly modified product construction.

**Construction 9 (complement & product construction).**

1. For finite positive constraints, see Construction 6.
2. For finite negative constraints, the constraint automaton is constructed from the deterministic automaton for $c$ by complement construction. For $\mathcal{A}_c = (A, S)$, let $\mathcal{A}_{\neg c} = (A', S)$ be the constraint automaton where $A'$ is the complement of $A$.
3. For finite conjunctive (disjunctive) constraints, the constraint automaton is constructed from the deterministic automata for the components by product construction. For simplicity, we describe the construction for the case of two constraints.

For constraint automata $\mathcal{A}_{c,i} = (A_i, S)$ for a graph grammar $(\mathcal{R}, S)$ with underlying finite automata $A_i = (\mathcal{C}_i, \mathcal{R}'_i, \delta_i, C_{0i}, F_i)$, let $\mathcal{A}_c = (A, S)$ be the constraint automaton with $A = (\mathcal{C}, \mathcal{R}', \delta, C_0, F)$ where $\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2$ is the product of $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{R}' = \{\langle p, \text{ac}_1 \wedge \text{ac}_2\rangle \mid \langle p, \text{ac}_i\rangle \in \mathcal{R}_i\}$, $C_0 = \{(c_{01}, c_{02}) \mid c_{0i} \in \mathcal{C}_{0i}\}$ is the set of initial states, and $F = \{(f_1, f_2) \mid f_i \in F_i\}$ is the set of final states in the case of conjunction and $F = \{(f_1, f_2) \mid f_1 \in F_1 \vee f_2 \in F_2\}$ in the case of disjunction. The transition function is given by $\delta((b_1, b_2), \varrho) = (\delta_1(b_1, \varrho_1), \delta_2(b_2, \varrho_2))$ where $b_i \in \mathcal{C}_i$, $\varrho_i = \langle p, \text{ac}_i\rangle$, and $\varrho = \langle p, \text{ac}_1 \wedge \text{ac}_2\rangle$.

**Example 23 (complement construction).** For the constraint ¬2tok, the constraint automaton $\mathcal{A}_{\neg\text{2tok}}$ is constructed from the deterministic automaton $\mathcal{A}'_{\text{2tok}}$ according to the complement construction. The result is given in Figure 3.11.

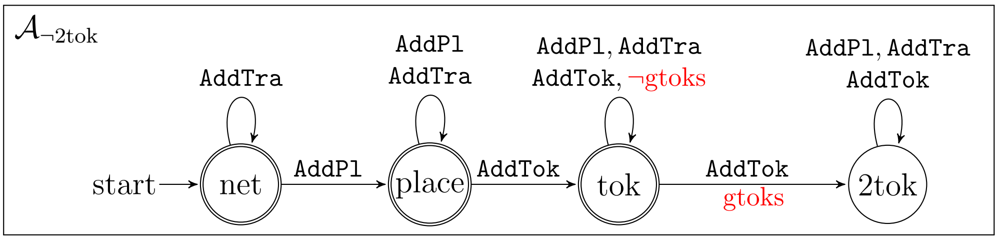


Figure 3.11: The complement automaton $\mathcal{A}_{\neg\text{2tok}}$.

**Example 24 (product construction).** Let $\mathcal{A}'_{\text{tok}}$ be the deterministic automaton obtained from the automaton $\mathcal{A}'_{\text{2tok}}$ of Example 21 by deleting the state 2tok and all incident edges (see Figure 3.10). For the constraint rtok = tok ∧ ¬2tok, the automaton $\mathcal{A}_{\text{rtok}}$ is constructed from the automata $\mathcal{A}'_{\text{tok}}$ and $\mathcal{A}'_{\neg\text{2tok}}$ in Figures 3.10 and 3.11 according to the product construction. The result is given in Figure 3.12.

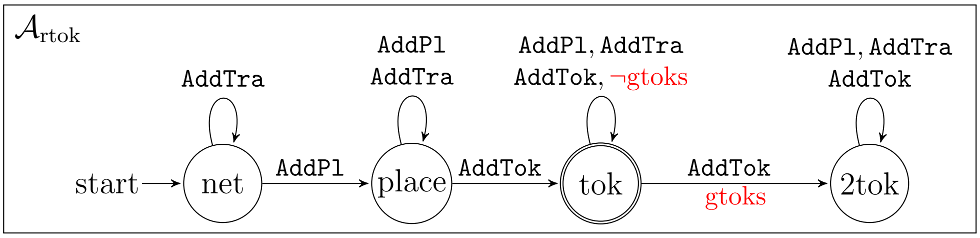


Figure 3.12: The product automaton $\mathcal{A}_{\text{rtok}}$.

Pairs $(b, b')$ with $b \sqsubseteq b'$ are represented by $b'$. Only states accessible from the initial state are drawn.

**Proof (of Lemma 14).** The statements follow immediately from Lemma 10 and the results on complement and product automata in formal language theory.
1. See the proof of Lemma 10.
2. Let $\mathcal{A}_c = (A, S)$ with deterministic automaton $A = (\mathcal{C}, \mathcal{R}', \rightarrow, C_0, F)$ and $\mathcal{A}_{\neg c} = (A', S)$ be the complement automaton with $A' = (\mathcal{C}, \mathcal{R}', \rightarrow, C_0, \mathcal{C} - F)$. By

construction, $b \rightarrow^* b'$ is a path in $\mathcal{A}'$ iff $b \rightarrow^* b'$ is a path in $\mathcal{A}$. As a consequence, we obtain the language equality.

$$\begin{array}{lll} & G \in L(\mathcal{A}_{\neg c}) & \\ \Leftrightarrow & \exists\, w \in L(A').S \Rightarrow_w^* G & (\text{Def } L(\mathcal{A}_{\neg c})) \\ \Leftrightarrow & \exists \text{ path } c_0 \rightarrow_w f \in \mathcal{C}{-}F.S \Rightarrow_w^* G & (\text{Def } L(\mathcal{A}')) \\ \Leftrightarrow & \exists \text{ path } c_0 \rightarrow_w f \in F.S \Rightarrow_w^* G & (\text{Def } L(A)) \\ \Leftrightarrow & \exists\, w' \in L(A).S \Rightarrow_w^* G & (\text{Def } L(\mathcal{A}_c)) \\ \Leftrightarrow & G \in L(GG) \cap \llbracket \neg c \rrbracket & (\text{Lemma 11}) \end{array}$$

where $c_0 \in C_0$. Thus, $L(A_{\neg c}) = L(GG) \cap \llbracket \neg c \rrbracket$.
3. Let $\mathcal{A}_{c_i} = (A_i, S)$ with $A_i = (\mathcal{C}, \mathcal{R}', \rightarrow_i, C_{0i}, F_i)$ be deterministic automata and $\mathcal{A}_c = (A, S)$ with $A = (\mathcal{C}, \mathcal{R}', \rightarrow, C_0, F)$ the product automaton. By construction (*) $\exists (b_1, b_2) \rightarrow_\varrho (b'_1, b'_2)$ in $\mathcal{A}_c$ iff $\exists b_i \rightarrow_{\varrho_i} b'_i$ in $\mathcal{A}_{c_i}$ $(i = 1, 2)$. As a consequence, we obtain the language equality.

$$\begin{array}{lll} & G \in L(\mathcal{A}_c) & \\ \Leftrightarrow & \exists\, w \in L(A).S \Rightarrow_w^* G & (\text{Def } L(\mathcal{A}_c)) \\ \Leftrightarrow & \exists \text{ path } (c_{01}, c_{02}) \rightarrow_w (c_1, c_2) \in F.S \Rightarrow_w^* G & (\text{Def } L(A)) \\ \Leftrightarrow & \exists \text{ paths } c_{0i} \rightarrow_{w_i} c_i \in F_i.S \Rightarrow_w^* G & (\text{Const 9, (*)}) \\ \Leftrightarrow & G \in L(GG) \cap \llbracket c_1 \rrbracket \text{ and } G \in L(GG) \cap \llbracket c_2 \rrbracket & (\text{Lemma 11}) \\ \Leftrightarrow & G \in L(GG) \cap \llbracket c_1 \wedge c_2 \rrbracket & (\text{Logic}) \end{array}$$

where $(c_{01}, c_{02}) \in C_0$, $c_{0i} \in C_{0i}$ $(i = 1, 2)$. Thus, $L(A_{c_1 \wedge c_2}){=}L(GG) \cap \llbracket c_1 \wedge c_2 \rrbracket$. □

## Filtering

We construct a graph grammar from a constraint automaton and derive our main theorem: the Filter Theorem for graph grammars and constraints.

**Lemma 15 (from constraint automata to graph grammars [Bec16]).** For every (goal-oriented) constraint automaton $\mathcal{A}_c$, a (goal-oriented) graph grammar $GG_c$ can be constructed effectively such that $L(GG_c) = L(\mathcal{A}_c)$.

**Construction 10.** For simplicity, we give the construction for deterministic automata. For non-deterministic automata, the construction is similar.
Let $\mathcal{A}_c = (A, S)$ be a deterministic constraint automaton with the underlying automaton $A = (\mathcal{C}, \mathcal{R}', \delta, C_0, \{c\})$. Then the graph grammar $GG_c = (\mathcal{C}, \mathcal{R}'_c, S)$ is constructed as follows. The nonterminals of the grammar are the constraints in $\mathcal{C}$.

The rule set $\mathcal{R}'_c$ is induced by the transition function $\delta$: for instruction $\delta(c_1, \varrho) = c_2$ with rule $\varrho = \langle L \Rightarrow R, \text{ac}\rangle$, we create a new rule $\langle L' \Rightarrow R', \text{ac}'\rangle$ where the states $c_1$ and $c_2$ are integrated into the left- and the right-hand side, respectively, and the constraint ac is shifted to $L'$. In more detail, let $\mathcal{R}'_c = \{S \Rightarrow S + \textcircled{c_0} \mid c_0 \in C_0\} \cup \{\langle c_1, \varrho, c_2\rangle \mid \delta(c_1, \varrho) = c_2\} \cup \{\ \textcircled{c} \ \Rightarrow \emptyset\}$ where the start rules add an initial state $c_0$, the simulating rules $\langle c_1, \varrho, c_2\rangle = \langle L + \textcircled{c_1} \Rightarrow R + \textcircled{c_2}, \text{ac}'\rangle$ with $\text{ac}' = \text{Shift}(L \hookrightarrow L + \textcircled{c_1}, \text{ac})$ simulate the working in the automaton, and the deleting rule allows to terminate.

**Example 25 (from automaton to grammar).** The graph grammar $GG_{\text{rtok}}$ can be derived from $\mathcal{A}_{\text{rtok}}$ in Figure 3.12. The states net, pl, tok, 2tok become the "nonterminal symbols" of the grammar, pl abbreviates place. The rules are as in Figure 3.13.

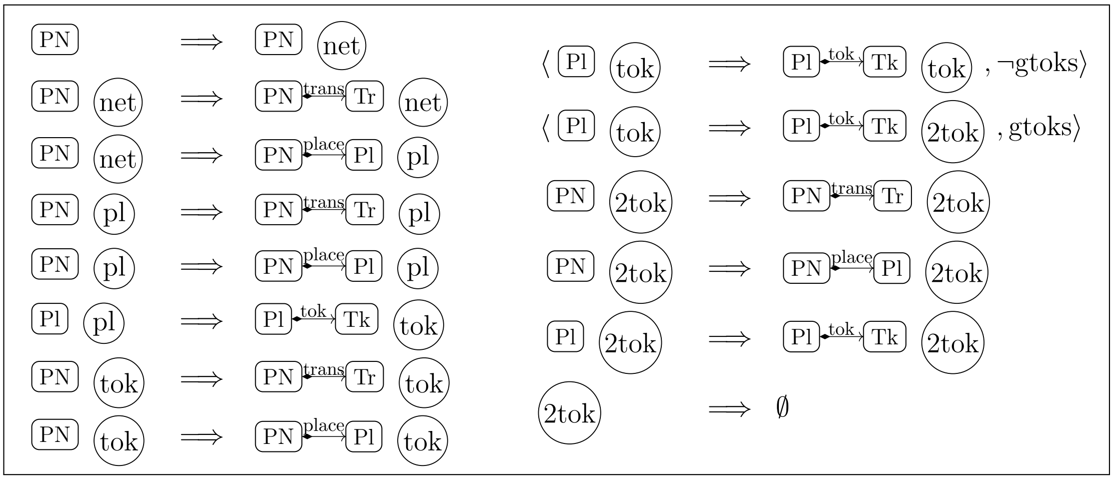


Figure 3.13: The graph grammar $GG_{\text{rtok}}$

**Remark.** Since $\mathcal{A}_c$ is a goal-oriented automaton, the resulting grammar is.

**Theorem 1 (Filter Theorem).** For arbitrary graph grammars $GG$, arbitrary graph constraints $c$, and for an index $i \geq 0$, goal-oriented graph grammars $GG_{c,i}$ ($GG_c$) can be constructed such that

1. $L(GG_{c,i}) \subseteq L(GG) \cap [\![c]\!]$ and
2. in case of termination, $L(GG_c) = L(GG) \cap [\![c]\!]$.

**Example 26.** For the graph grammar for Petri nets (Example 1) and the constraint rtok, the goal-oriented grammar $GG_{\mathrm{rtok}}$ is shown in Figure 3.13.

**Proof.** The theorem follows immediately from Lemmata 10, 12, and 15. □

**Remark (typed graphs with containment).** All results in this chapter can be obtained for typed graphs with containment (see Definition 3): For the construction of existential weakest liberal preconditions, $\mathcal{E}'$-$\mathcal{M}$ pair factorization and the existence of $\mathcal{M}$-pushout is used. Typed graphs with containment and morphisms form a category (Lemma 1), that has these properties where $\mathcal{E}'$ and $\mathcal{M}$ are the classes of all jointly surjective pairs of all injective morphisms, respectively.

## 3.3 Related work

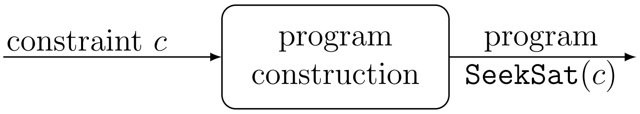
In this section, we present some related concepts on model generation and the integration of constraints in chronological order. For logic-based approaches and a comparison with the graph-based approaches, see Radke et al. [RAB+18].

### Model generation

Most approaches to instance generation are *logic-oriented*, e.g., Cabot et al. [CCR07], Kuhlmann and Gogolla [KG12]. They translate class models with OCL constraints into logical facts and formulas, such as Alloy [Jac12]. Then, an instance can be generated or it can be shown that no instances exist.

Alternatively, *graph grammars* have been shown to be suitable and natural to specify (domain-specific) visual languages in a constructive way. For an overview, see Bardohl et al. [BMST99].

In **Pennemann** [Pen09], an algorithm is given that generates for each graph condition $c$ a non-deterministic program $\texttt{SeekSat}(c)$, which finds a valid graph for every satisfiable condition. Starting from the empty graph, the algorithm adds items, progressing to a valid graph that satisfies the constraint. Since negative conditions are refuted, the program needs backtracking. The algorithm is correct and complete, but it is not guaranteed to terminate in general. For conditions of the form $\exists\,(a, c)$ where $c = \texttt{true}$, $\texttt{SeekSat}$ is guaranteed to terminate.



In **Arendt et al.** [AHRT14], **Radke et al.** [RAB+15, RAB+18], a translation of OCL constraints to graph patterns or graph constraints is presented and therefore bridge between both approaches. To formally treat models and meta-models (without OCL constraints) they are translated to instance and type graphs. Hence, they follow the graph-based approach keeping the graph structure of models as units of abstraction where graph axioms are satisfied by default. Meanwhile, Bergmann [Ber14] has implemented a translator of OCL constraints to graph patterns, which is rather an efficient implementation, than a formal translation.

In **Schneider et al.** [SLO17, SLO18], a parallelizable symbolic model generation approach for attributed graphs is presented. Firstly, a new logic for attributed graph properties is introduced, where the graph and the attribution part are separated. The graph part is equivalent to first-order logic on graphs, the attribution part is added to this graph part by reverting to the symbolic approach to graph attribution, where attributes are represented symbolically by variables whose possible values are specified by a set of constraints making use of algebraic specifications. Secondly, the parallelizable algorithm gradually generates a finite set of so-called symbolic models, where each symbolic model describes a set of finite graphs satisfying the graph property. The set of symbolic models jointly describes all finite models for the graph property and does not describe any finite graph violating the property. The attribution part uses an oracle, allowing for flexible adoption of different SMT (Satisfiability Modulo Theories) solvers, in particular, the SMT solver Z3. The algorithm is implemented in the tool `AutoGraph`.

In **Semeráth et al. 2018** [SNV18] and **Varró et al. 2018**, [VSSH18], instances of meta-models are generated with a graph solver by refining a general model with a finite sequence of graph transformation rules along a refinement relation, yielding a concrete instance. Partial instance models are defined, where a 3-valued logical value is assigned to each element of the graph, extending a boolean value by uncertainty. In each refinement step, it is checked, if the constraint is violated. In **Semeráth et al. 2020** [SBL$^{+}$20], this approach is extended to generate models with structural and attribute constraints. They combine the structural graph solver with an SMT (Satisfiability Modulo Theories) solver. The numerical part of the constraints is constructed as a conjunction of numerical clauses and solved by the SMT-solver.

In **Nassar et al.** [NKAT20], a rule-based, configurable approach to automate model generation, which produces instance models of meta-models with multiplicity constraints conforming to the Eclipse Modeling Framework, is presented. The repair approach by the author [NKR17] is used for the generation of large EMF models, with up to half a million graph elements.

## Integration of constraints

In the following, for positive constraints, we compare the construction by Radke et al. [RAB$^+$15, RAB$^+$18] with the backward construction. It turns out that the backward construction is finer that the construction of Radke et al. [RAB$^+$18].

In **Radke et al.** [RAB$^+$15, RAB$^+$18], given a positive constraint, the integration of constraints is done by replacing all rules by the corresponding guaranteeing rules $\text{gua}(\varrho, c)$. This guarantees that for all derivations via $c$-guaranteeing rules, the result satisfies the constraint $c$.

The *application-condition-based* approach of Radke et al. is very simple. The problem is that in several cases, the language $L(GG_c)$ is empty (see, e.g., Example 27). In practical applications, no output is only acceptable if the set of graphs satisfying $c$ in the language $L(GG)$ is empty.

**Fact 4 (refining construction).** For positive constraint, the backward construction is finer than the one of Radke et al., i.e., for each solution of the weak filter problem by Radke et al. there is a solution by the backward construction. The converse direction does not hold.

**Sketch of proof.** For positive constraints $c$, $b$ in $\text{Wlp}_\exists(\varrho, c)$ implies $b$ contains $c$, the application condition $\text{Gua}(\varrho, b)$ contains $\text{Gua}(\varrho, c)$ (in the sense of Definition 11), and whenever the rule $\text{gua}(\varrho, c)$ is applicable, then the rule $\text{gua}(\varrho, b)$ is applicable. Thus, every solution by Radke et al. implies a solution by the backward construction.

Example 27 shows, that the converse does not hold.

**Example 27 (Refining construction).** Consider the graph grammar $GG$ in Example 1 with the start graph $S = \boxed{\text{PN}} \xrightarrow{\text{place}} \boxed{\text{Pl}}$. For the constraint $c = \text{tok}$, meaning there is a place with one token, the language $L(GG_c)$ of Radke et al. is infinite. For the constraint $c = 2\text{tok}$, meaning there is a place with two tokens, the language $L(GG_c)$ of Radke et al. is empty. The reason is that the start graph $S$ has a place without token and the $c$-guaranteeing rule of `AddTok` is not applicable to $S$ because `AddTok` adds only one token.

In contrast, the backward construction terminates and leads the to constraint automaton $\mathcal{A}_{2\text{tok}}$ from Figure 3.5 and, from that, to a graph grammar $GG_c$ with $L(GG_c) = L(GG) \cap [\![c]\!]$.

In **Becker** [Bec16, Thm 1], it is shown that the filter problem is solvable for arbitrary graph grammars $GG$ and arbitrary graph constraints: The used method is generate & test: first, generate a graph and, then, test whether the constraint is satisfied. Unfortunately, the grammar is not goal-oriented. If the final graph does not satisfy the constraint, the construction tries to find a graph satisfying the constraint again. In Becker [Bec16, Thm 2 & 3], it is shown that the filter problem is solvable by a goal-oriented grammar. The statement is based on a backward construction similar to ours. [Bec16, Algorithm 1] assumes the existence of a "disjunctive normal form" for constraints. In general, for arbitrary constraints, there is only a normal form. Algorithm 1 makes use of equivalence and implication, i.e., it is not effective. In this chapter, we use minimization and containment which can be easily constructed and checked and implies equivalence and implication, respectively.

| | grammar | constraint | rel | method | uses |
|---|---|---|---|---|---|
| Becker | arbitrary | arbitrary | $=$ | generate & test | gua |
| Radke et al. | arbitrary | arbitrary | $\subseteq$ | appl conditions | gua |
| this chapter | arbitrary | arbitrary | $\subseteq$ | automata-based | $\mathrm{Wlp}_\exists$, gua |
| this chapter | non-del & bp | positive | $=$ | automata-based | $\mathrm{Wlp}_\exists$, gua |

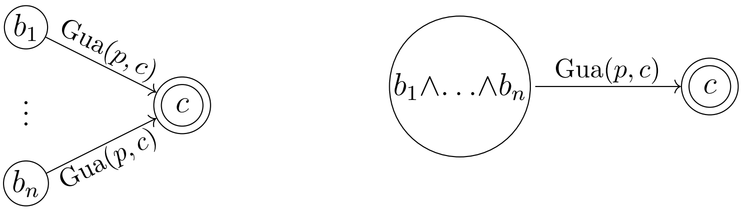
where non-del stands for plain, non-deleting and bp for bounded path

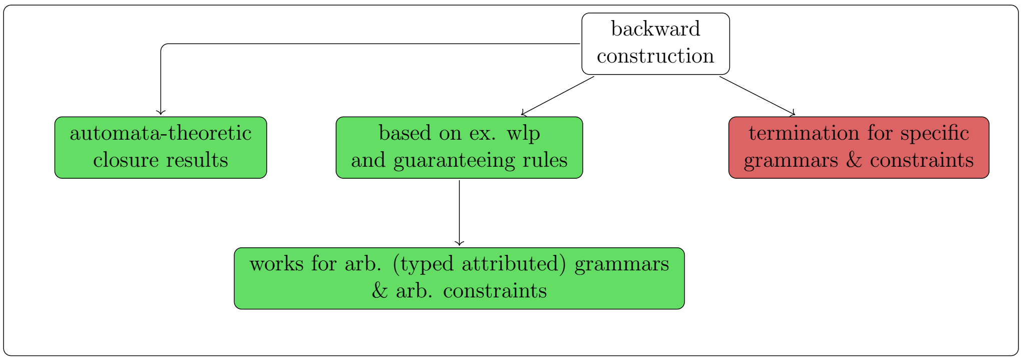
- The application-condition-based constructions of Becker and Radke et al. are simple: Becker uses only one *c*-guaranteeing rule for the final test; Radke et al. equip all rules with a *c*-guaranteeing application condition.
- The automata-based construction is more difficult: One has to construct the least weakest preconditions ($\mathrm{Wlp}_\exists$) as well as *c*-guaranteeing rules (gua).

In **Hildebrandt et al.** [HLBG12], an approach to integrate constraints into model transformations based on triple graph grammars is presented. In addition to the integration, the approach can check if the grammar is "forward (backward) valid", meaning that a transformation always produces a model satisfying the constraints, provided that the input model satisfies the constraints. The invariant checker either reports that a constraint is preserved for a given set of rules, or it automatically calculates all minimal situations indicating why rules might be applied to a constraint-satisfying graph leading to a graph, violating the constraint.

## 3.4 Conclusion

The backward construction works for arbitrary graph grammars and arbitrary constraints. If the existential weakest liberal precondition $\mathrm{Wlp}_\exists(\varrho, c) = \vee_{i\in I} b_i$ is a disjunction of several constraints, there is a proper decomposition into the smaller components $b_i$ which can be handled in the same way. For plain, increasing grammars and positive constraints, $\mathrm{Wlp}_\exists(p, d)$ is a disjunction of positive constraints and the states in constraint automata are positive constraints. If $\mathrm{Wlp}_\exists(\varrho, c)$ is not a disjunction of several constraints, e.g., $\wedge_{i \in I} b_i$, then the complex constraint $\wedge_{i \in I} b_i$ has to be handled.




backward construction

automata-theoretic closure results

based on ex. wlp and guaranteeing rules

termination for specific grammars & constraints

works for arb. (typed attributed) grammars & arb. constraints

Figure 3.14: Advantages and disadvantages of the backward construction

To summarize (see Figure 3.14), the backward construction

(1) **works for arbitrary graph grammars and constraints.** the backward construction is based on the construction of existential weakest liberal preconditions and guaranteeing rules. These constructions can be done for arbitrary (typed attributed) graph grammars and constraints.

(2) **has automata-theoretic closure results.** Similar to formal language theory, there is a transformation of (nondeterministic) constraint automata into deterministic ones. Moreover, constraint automata are closed under the Boolean operations complementation and product. In this way, we can construct constraint automata for more complex constraints. (We construct constraint automata for the basic constraints, use the Boolean operations for the constraint automata and obtain a constraint automaton for the more complex constraint.)

The drawback of the construction is the termination for specific grammars and constraints.

## Further topics

**Program automata.** For constraints of the form $\forall(a, c)$, e.g., the constraint alltok $= \forall(\,\boxed{\text{Pl}}\,, \exists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,)$, meaning that all Pl-nodes have a tok-edge to a Tk-node, the constraint automata can be generalized to program automata, where the transitions of the automata are decorated by graph programs instead of rules equipped with application conditions. This way, a finite program automaton may be obtained instead of an "infinite" constraint automaton. This may be seen as the motivation for the next chapter, where we consider so-called repair programs.

# Chapter 4

# Graph repair

In model-driven software engineering, the primary artifacts are models (Sendall et al. [SK03], Heckel and Taentzer [HT20]). Models have to be consistent with respect to a set of constraints, specified for example in the Object Constraint Language (OCL) [Obj14]. To increase the productivity of software development, it is necessary to automatically detect and resolve inconsistencies arising during the development process, called model repair (see, e.g., Nentwich et al. [NEF03], Macedo et al. [MTC17], Nassar et al. [NRA17]). Our approach to model repair is the following.

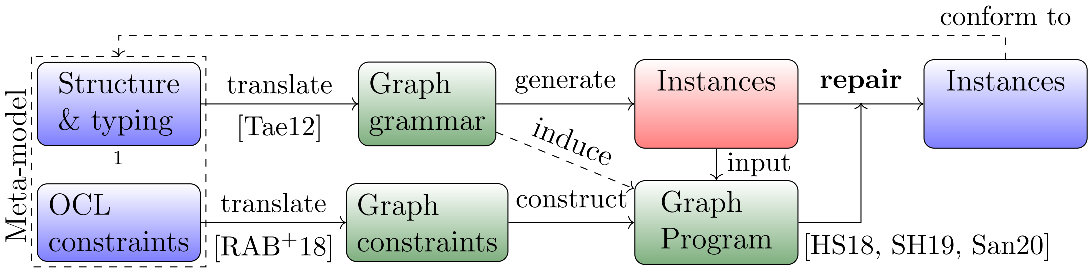


Figure 4.1: Instance repair of meta-models

(1) Translate the structure of the meta-model and its typing into a (typed) graph grammar $GG$ (Taentzer [Tae12]) and OCL constraints into graph constraints $c$ (Radke et al. [RAB$^+$18]).

(2) Generate instances $I \in L(GG)$ in the language of the constructed grammar, possibly violating the constraint (drawn in red in Figure 4.1).

(3) Repair instances to instances $I' \in L(GG)$ satisfying graph constraints $c$.

[1] The illustration shows the main components of a meta-model: the structure and the typing as well as OCL constraints. The OCL constraints are typed over the meta-model. The relations between the structure and typing of the meta-model and the OCL constraints are not drawn. For a definition of a meta-model, see Section 5.1.

Up to now, a well-founded theory to repair arbitrary instances into instances conforming to the constraints is missing.

**Bibliographic notes.** In Taentzer [Tae12], a (restricted) form of meta-models is formalized by type graphs with multiplicities. From these meta-models, an instance-generating (typed) graph grammar is introduced for creating typed graphs representing the structures of the models. It is shown that for each type graph with inheritance and multiplicities an instance-generating (typed) graph grammar can be constructed, such that the languages are equal. In Radke et al. [RAB$^+$18], a meta-model is translated to a type graph and so-called Essential OCL invariants are translated to graph constraints. It is shown that a model satisfies an Essential OCL invariant iff its corresponding typed graph satisfies the typed graph condition.

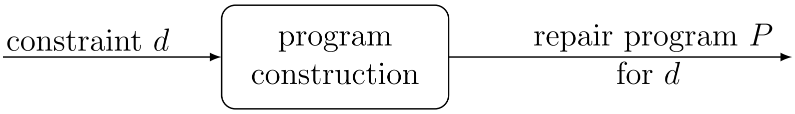
Since instances can be modelled as graphs, in this chapter, we consider the problem of *graph repair*: Given a graph and a constraint, try to construct a graph, satisfying the constraint. To get a graph repair for a graph and a constraint, we construct *repair programs.* A repair program is a graph program with the property that for every graph and every constraint,

(1) **Existence.** There exists a transformation with the program and
(2) **Correctness.** All transformations via the program yield a graph satisfying the constraint.

This has the advantage, that we may create a repair program only once and can reuse it for every graph, provided that the constraints do not change.

**Repair problem**

**Given:** A graph constraint $d$.
**Task:** Try to find a repair program $P$: $\forall G \exists\, G \Rightarrow_P H$ and $\forall G \Rightarrow_P H.\ H \models d$.



We solve the repair problem for suitable constraints. The construction is done in two steps.

(1) Construct a repair program $P$ directly from the constraint such that for all graphs, there exists a transformation with the program and, for each application, the resulting graph satisfies the constraint.
(2) Apply the program $P$ to a graph $G$ to get a graph repair, that is, the resulting graph satisfies the constraint.

In Section 4.1, we introduce repair programs. In Section 4.2, we start with basic conditions, requiring the existence (non-existence) of a real morphism. In Section 4.3, we present repair programs for proper conditions. In Section 4.4, we show that there are repair programs for several conjunctive conditions. In Section 4.4, we show that there are repair programs for disjunctive conditions, provided that there exist repair programs for the subconditions. In Section 4.5, the repair results of the previous sections are summarized to a repair result for so-called legit conditions. The class of legit conditions are all proper or generalized proper[2] ones, preserving conjunctions of conditions, and all disjunctive conditions. In Section 4.6, we show that the constructed repair programs are stable, maximally preserving, and terminating. In Section 4.7, we consider grammar-based programs, i.e., programs which are based on a set of rules of a grammar, and grammar-based graph repair. In Section 4.8, we present some related concepts and compare them with our approach. In Section 4.9, we give a conclusion.

**Bibliographic notes.** The notion of repair programs is oriented at other notions of correct programs, as in Apt and Olderog [AO91]: A program is correct, if the application of the program yields a graph satisfying the constraint. A program is weak total correct if, additionally, there is no infinite transformation with transformation. It is total correct, if there exists a transformation with program. For repair programs, we require the correctness and the existence. In Nassar et al. [NRA17] a rule-based approach is presented, which is specialized for models, which are based on the Eclipse Modeling Framework, and works with restrictions on constraints and the input instances.

## 4.1 Repair programs

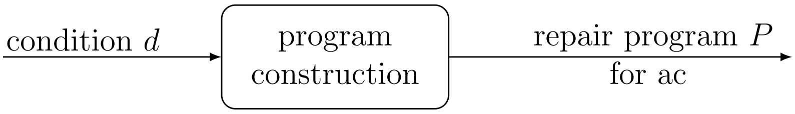
In this section, we define repair programs and sketch some desirable properties for these repair programs, which are highly relevant for our application to metamodeling and are based on the paper of Macedo et al. 2017 [MTC17].

A repair program for a constraint is a program such that for all graphs there exists a transformation and for every application to a graph, the resulting graph satisfies the constraint. More generally, a repair program for a condition over a graph $A$ is a program $P$ with interface $A$ such that for every triple $\langle g, h, i \rangle$ in the semantics of $P$, the composition of the interface relation $i$ and the morphism $h$ satisfies the condition. Similar to Schneider et al. [SLO19], one may construct a repair with respect to one graph: Given a constraint and a graph, construct a graph, satisfying

[2] Generalized proper: conditions obtained by a proper condition by replacing a subcondition with a condition possessing a repair program for the subcondition.

the constraint, based on the input graph. In contrast, the repair programs, have the properties for every graph. We have chosen this approach, because we may create a repair program only once and can reuse it for every graph. For a more detailed comparison with the approach of Schneider et al., we refer to Section 4.8.

**Definition 18 (repair programs).** A (typed) program $P$ is a *(typed) repair program* for a constraint $d$ if, for all (typed) graphs $G$, $\exists\, G \Rightarrow_P H$ and $\forall\, G \Rightarrow_P H$, $H \models d$.



A (typed) program with interface $A$ is an *$A$-program* if the interface relation $i$ of the (typed) program is total, and the codomain of $i$ is $A$. An $A$-program $P$ is a *repair program* for a condition ac over $A$, if for all morphisms $g\colon A \hookrightarrow G$, $\exists\, g \Rightarrow_{P,i} h$ and $\forall\, g \Rightarrow_{P,i} h$, $h \circ i \models$ ac.

$$\begin{tikzcd}
\overset{d}{\rhd} & A \arrow[r, "i"] \arrow[d, "g"'] & A \arrow[d, "h"] \\
 & G \arrow[r, Rightarrow, "P"'] & H
\end{tikzcd}$$

**Example 28.** For the constraint $d = \nexists(\ \boxed{\text{Pl}} \overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}}\ )$, meaning there do not exist two tok-edges between a Pl-node and a Tk-node, the program

$$P_d = \langle\ \boxed{\text{Pl}} \overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}} \ \Rightarrow\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ \rangle{\downarrow}$$

is a repair program for $d$. Given an input graph, whenever there are two parallel tok-edges, the program deletes one. This deletion of one of the edges is done as long as possible. Since $d$ is a constraint, i.e., the domain is the empty graph, the interfaces of the program are empty.

Given a condition, there may be several different repair programs for it. A program for a condition is *destructive*, if it deletes the input graph and creates a graph satisfying the condition from the empty graph. In Pennemann [Pen08], a non-deterministic, increasing algorithm `SeekSat` for the solution of the satisfiability problem is presented. The algorithm returns a graph for every satisfiable condition. The algorithm is correct and complete but is in general not guaranteed to terminate.

**Lemma 16.** For every satisfiable[3] condition, there exists a destructive repair program.

**Proof.** For a satisfiable condition $d$, the program $P_d = \langle \texttt{Delete}\downarrow; \texttt{SeekSat}(d)\rangle$, where `Delete` denotes the program to delete an arbitrary graph (with arbitrary types), is a destructive repair program for $d$. □

We are interested in so-called stable, maximally preserving, and terminating repair programs. Informally, a repair program is

(1) *stable*, if it does nothing whenever the condition is already satisfied.

(2) *m*aximally preserving , if, informally, items are preserved whenever possible,

(3) *terminating*, if there is no infinite transformation with the repair program.

Formal definitions are given in Section 4.6.

**Example 29.** The repair program from Example 28 is (1) stable, (2) maximally preserving, and terminating. (1) The program $P_d$ for $d$ can only delete one of the two parallel tok-edges. This is done as long as possible. Whenever all occurrences of the two parallel tok-edges are deleted, the program cannot be applied. Consequently, it does not change the input graph, provided that the condition is satisfied, i.e., it is stable. (2) Whenever the graph does not satisfy the constraint $d$, the constraint cannot be repaired without deleting at least one of the two parallel tok-edges, for each occurrence in the graph. The repair program $P_d$ for $d$ deletes only one of the tok-edges, as long as possible. For an input graph, the program preserves as much of the edges, as possible, i.e., the program is maximally preserving. (3) The program is decreasing, and, consequently there is no infinite transformation, i.e., it is terminating,

In the remainder of the chapter, we construct repair programs for different kinds of constraints or - more generally - conditions. For this purpose, we start with small conditions, so-called basic conditions, requiring the existence (non-existence) of a real morphism. For larger conditions, the repair program is composed of basic repair programs.

**Bibliographic notes.** One may require termination for all the repair programs. In contrast, we have shown it as a special requirement. The reason for this, is that the construction of terminating repair programs is very difficult, and we only get terminating repair programs, for specific constraints and specific rule sets. For example, the destructive repair program may not be terminating, in general.

[3] A condition $c$ is *satisfiable* if there is a morphism $p$ that satisfies $c$.

## 4.2 Basic conditions

In the following, we construct repair programs for “smallest” conditions. For this purpose, we start with basic conditions, i.e., conditions of the form $\exists\, a$ or $\nexists\, a$ with real morphism $a\colon A \hookrightarrow C$ requiring the existence or non-existence of a morphism.

**Definition 19 (basic condition).** Given a real morphism $a\colon A \hookrightarrow C$ with $A \subset C$, a condition is *basic* if it is of the form $\exists\, a$ or $\nexists\, a$. A basic condition of the form $\exists\, a$ ($\nexists\, a$) is *positive* (*negative*).

For basic conditions, we construct so-called (1) *repairing* sets from the morphism of the condition and (2) repair programs based on the repairing sets using the try statement and the “as long as possible” iteration, respectively.

**Lemma 17 (basic repair).** For basic conditions, there are repair programs.

**Proof.** For the proof, see the proof of Theorem 2. □

In the following, we present a so-called ad hoc construction and a solid construction. Both constructions are based on the same idea. For basic conditions, we decompose a real morphism $a\colon A \hookrightarrow C$ into subgraphs $B$ of $C$, such that $A \hookrightarrow B \subset C$. These subgraphs form the building blocks of the resulting rules: for positive conditions $\exists\, a$, we construct the rules $B \Rightarrow C$, for negative conditions $\nexists\, a$, the rules $C \Rightarrow B$. For positive conditions, both constructions require that the condition $\exists\, a$ shifted to the left-hand side of the rule, is not satisfied. This guarantees stability. These rule sets are equipped with interfaces. The ad hoc construction allows taking a superset of $C$ as the right-hand side of the rule. This allows us to be more flexible with the repairs. The superset may be the result of an arbitrary application of rule sets for adding nodes and edges. The solid construction requires an additional application condition, which guarantees that the rule with the maximal subgraph $B$ of $C$ is applied. This guarantees termination. For negative conditions, the solid construction requires that the rule only deletes one edge, if this is possible, or one node, otherwise. This guarantees maximal preservation.

Given an input morphism $g\colon A \hookrightarrow G$, and a condition $\exists\, a$ that is not satisfied, the construction extends any occurrence of $B$ in the graph $G$ to an occurrence of $C$. For a condition $\nexists\, a$ that is not satisfied, the construction removes any occurrence of $C$ in the graph $G$ to an occurrence of $B$. By choosing the maximal $B$, application of the rule $B \Rightarrow C$ ($C \Rightarrow B$) yields minimal additions (deletions).

$$\begin{tikzcd}
& A \arrow[dl, hook', "b"'] \arrow[dr, hook, "a"] \arrow[ddl, bend right=60, hook', "g"'] \arrow[ddr, bend left=60, hook, "h"] & \\
B \arrow[rr, Rightarrow] \arrow[d, hook, "g'"'] & & C \arrow[d, hook, "h'"] \\
G \arrow[rr, Rightarrow] \arrow[from=2-1, to=3-3, phantom, "="] & & H
\end{tikzcd}$$

We will start with the ad hoc construction, which is easier to construct than the solid construction.

**Construction 11 (ad hoc repair).** For a real morphism $a\colon A \hookrightarrow C$ with $A \subset C$, the programs are as follows.

$$P_{\exists a} = \texttt{try}\ \widehat{\mathcal{R}_a} \quad \text{with } \widehat{\mathcal{R}_a} = \{\langle b, B \Rightarrow \widehat{C}, \text{ac}, \widehat{a}\rangle \mid A \hookrightarrow^b B \subset C \subseteq \widehat{C}\}, \text{ and}$$
$$P_{\nexists a} = \widehat{\mathcal{S}_a}'\!\downarrow \quad \text{with } \widehat{\mathcal{S}_a} = \{\langle a, C \Rightarrow B, b\rangle \mid A \hookrightarrow^b B \subset C\},$$

where $\widehat{a}\colon A \hookrightarrow \widehat{C}$, ac $=$ Shift$(b, \nexists a)$, and $'$ denotes the dangling-edges operator.

**Example 30.** Consider the constraint $d = \exists a$ with $a\colon \emptyset \hookrightarrow \boxed{\text{Pl}}$ meaning, that there exists Pl-node. There is one graph $B = \emptyset$ such that $\emptyset \subseteq B \subset \boxed{\text{Pl}}$. Application of the ad hoc construction may yield a rule set $\widehat{\mathcal{R}_a}$ with the rules

$$\langle \emptyset \ \Rightarrow \ \boxed{\text{Pl}}, \nexists \boxed{\text{Pl}}\,\rangle$$

The application conditions are constructed by shifting the condition $\nexists a$ to the left-hand side of the rule. Since the ad hoc construction allows for a super set of the Pl-node, the ad hoc construction may extend the rule set $\widehat{\mathcal{R}_a}$ by a second rule:

$$\langle \emptyset \ \Rightarrow \ \boxed{\text{Pl}}\ \boxed{\text{Pl}}, \nexists \boxed{\text{Pl}}\,\rangle$$

The application of the first rule yields a graph with one Pl-node, the second one with two Pl-nodes. The program $P_d = \texttt{try}\ \widehat{\mathcal{R}_a}$ is a repair program for $d$.

**Example 31.** Consider the condition $\nexists a$ with $a\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$. By the ad hoc construction, the rule set

$$\widehat{\mathcal{S}_a} = \{\langle a, \ \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \ \Rightarrow \ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,, b\rangle\}$$

with $b\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$ constitutes the repairing set for $\nexists a$. The rule is node-deleting, i.e., it deletes a Tk-node with an incident edge. For the application of the rule to a graph, it may be necessary to delete the dangling edges: Application of the rule deletes a Tk-node and a tok-edge. If the occurrence of this Tk-node in the application graph possesses dangling edges, the dangling edges are deleted, as well. The program $P_d = \widehat{\mathcal{S}_a}'\!\downarrow$ is a repair program for $d$.

The ad hoc is easy to construct, but may be, in general, not terminating or maximally preserving. The solid construction is more restrictive, but is stable, maximally preserving, and terminating.

**Construction 12 (solid repair).** For a real morphism $a\colon A \hookrightarrow C$, the programs are as follows.

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

where $\mathrm{ac} = \mathrm{Shift}(A \hookrightarrow B, \nexists a)$, $\mathrm{ac}_B = \bigwedge_{B'} \nexists B'$, $\bigwedge_{B'}$ ranges over $B'$ with $B \subset B' \subseteq C$, (mp)[4] $\texttt{if}\ \mathrm{E}_C \supset \mathrm{E}_A\ \texttt{then}\ |\mathrm{V}_C| = |\mathrm{V}_B|, |\mathrm{E}_C| = |\mathrm{E}_B| + 1\ \texttt{else}\ |\mathrm{V}_C| = |\mathrm{V}_B| + 1$, and $'$ denotes the dangling-edges operator.

**Convention.** In the following, repair programs based on the solid (ad hoc) construction are said to be *solid* (*ad hoc*) repair programs.

**Informal Description.** The rules in $\mathcal{R}_a$ are of the form $B \Rightarrow C$ equipped with interfaces. They possess an application condition requiring the condition $\nexists a$, shifted from $A$ to $B$. By the application condition, each rule can only be applied iff the condition is not satisfied. The rules in $\mathcal{S}_a$ are decreasing and of the form $C \Rightarrow B$ where $A \subseteq B \subset C$. By $B \subset C$, both rule sets do not contain identical rules. The rules in $\mathcal{R}_a$ are equipped with an application condition requiring that no larger subgraph $B'$ of $C$ occurs. This yields termination: by the application condition, each rule can only be applied if no other rule whose left-hand side includes $B$ and is larger can be applied. The rules in $\mathcal{S}_a$ are restricted, such that, if the number of edges in $C$ is larger than the one in $A$, they delete one edge and no node, and delete a node, otherwise. We delete edges instead of nodes, whenever possible since it is more costly to delete nodes than edges.

**Fact 5.** Each solid repair program is an ad hoc repair program. The converse does not hold. The ad hoc repair programs are stable, but, in general, not maximally preserving and terminating (see Facts 14 and 15). The solid repair programs turn out to be stable, maximally preserving, and terminating (see Lemmata 23, 24, and 26).

A morphism $a\colon A \hookrightarrow C$ is *edge-increasing* if the number of edges in the codomain is larger than the number of edges in the domain, i.e., $|E_C| > |E_A|$. For edge-increasing morphisms, the rule set $\mathcal{S}_a$ and the program $\mathcal{S}'_a$ are equivalent.

[4] This requirement guarantees, that the resulting repair programs are maximally preserving. Therefore, we name it (mp).

**Fact 6.** For edge-increasing morphisms $a$, $\mathcal{S}_a \equiv \mathcal{S}'_a$.

**Example 32.** Consider the condition $d = \exists\, a$ with $a\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$, meaning that whenever there is a Pl-node there exists a Tk-node and a connecting containment tok-edge. There are two graphs $B_1 = \boxed{\text{Pl}}$ and $B_2 = \boxed{\text{Pl}}\ \boxed{\text{Tk}}$, such that $\boxed{\text{Pl}} \subseteq B_i \subset \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$. Application of the solid construction yields a rule set $\mathcal{R}_a$ with two rules.

$$\mathcal{R}_a = \begin{cases} \langle\, b_1,\ \boxed{\text{Pl}} & \Rightarrow\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}},\ \nexists\ \boxed{\text{Pl}}\ \ \boxed{\text{Tk}},\ a\, \rangle \\ \langle\, b_2,\ \boxed{\text{Pl}}\ \boxed{\text{Tk}} & \Rightarrow\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}},\ \nexists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \wedge \nexists\ \boxed{\text{Tk}}\ \ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}},\ a\, \rangle \end{cases}$$

where the left interface morphisms are $b_1\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}$ and $b_2\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}\ \boxed{\text{Tk}}$ and the right interface morphisms are the morphism $a$ of the condition. The application condition is constructed by shifting the condition $\nexists\, a$ over the left interface morphisms $b_i$ of the rule. For the first rule, there is a larger graph $B'_1 = \boxed{\text{Pl}}\ \ \boxed{\text{Tk}}$ such that $\boxed{\text{Pl}} \subset B'_1 \subseteq \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$. The first rule is equipped by a left application condition forbidding the existence of the graph $B'_1$. This yields termination, by the application condition, the rule is only applicable if the input graph does not have a Tk-node. The rule $\varrho_1$ requires a Pl-node and attaches a Tk-node and a connecting containment tok-edge, provided that there do not exist a Pl-node and a Tk-node. The second rule $\varrho_2$ requires an occurrence of a Pl- and a Tk-node and inserts a connecting containment tok-edge, provided there is no containment tok-edge from the occurrence of the Pl-node to the image of Tk-node, and there is no containment tok-edge to another Tk-node. The program $P_d = \texttt{try}\ \mathcal{R}_a$ is a repair program for $d$.

**Example 33.** Consider the condition $\nexists\, a$ with $a\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$. The morphism is edge-increasing and there is a graph $B_1 = \boxed{\text{Tk}}\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$ resulting in a rule which deletes exactly one edge. By the solid construction, the rule set

$$\mathcal{S}_a = \{\langle a,\ \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \ \Rightarrow\ \boxed{\text{Tk}}\ \ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}},\ b\rangle\}$$

with $b\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Tk}}\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$ constitutes the repairing set for $\nexists\, a$. The program $P_d = \mathcal{S}'_a\!\downarrow$ is a repair program for $d = \nexists\, a$.

**Bibliographic notes.** The repair programs are partial correct, in the sense of Apt and Olderog [AO91], w.r.t. $\texttt{true}$ and $d$. We will show, that the solid repair programs are terminating (see Lemma 26). From this it follows, that the solid repair programs are weak partial correct w.r.t. $\texttt{true}, d$. For all repair programs there exists a transformation with the program. Consequently, the solid repair programs are total correct w.r.t $\texttt{true}, d$.

## 4.3 Proper conditions

In this section, we construct repair programs for so-called proper conditions. They are more general than basic condition, i.e., proper conditions extend the basic conditions to the nesting structure of conditions without conjunctions and disjunctions.

A proper condition is a linear condition with alternating quantifiers ending with `true`. Additionally, some specific linear conditions with alternating quantifiers ending with `false` are said to be proper: conditions of the form $\forall(b, \texttt{false})$ equivalent to $\nexists\, b$, and conditions of the form $\exists\,(a, \forall(b, \texttt{false}))$ equivalent to $\exists\,(a, \nexists\, b)$.

**Definition 20 (proper conditions).** A linear condition in alternating normal form is *proper* if that ends with `true` or it is a condition of the form $\exists\,(a, \nexists\, b)$ or $\nexists\, b$. A proper condition of the form $\forall(a, c)$ (and $\exists\,(a, c)$) that ends with `true` is *universal* (and *existential*), respectively.

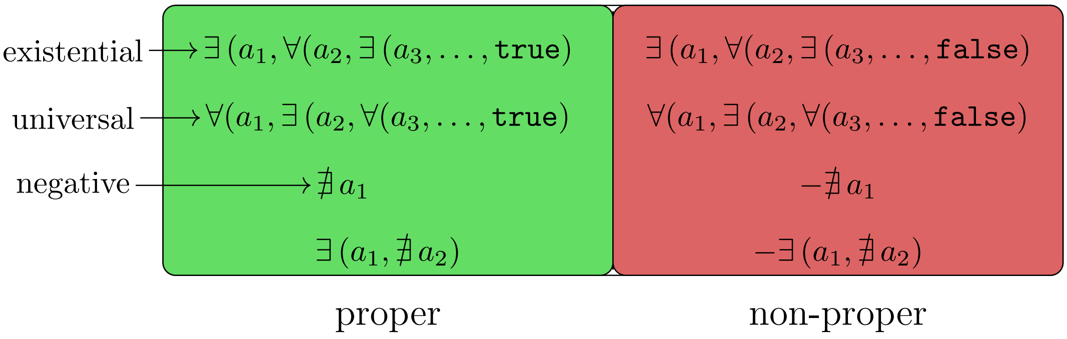

existential
universal
negative
$\exists\,(a_1, \forall(a_2, \exists\,(a_3, \dots, \texttt{true})$
$\forall(a_1, \exists\,(a_2, \forall(a_3, \dots, \texttt{true})$
$\nexists\, a_1$
$\exists\,(a_1, \nexists\, a_2)$
$\exists\,(a_1, \forall(a_2, \exists\,(a_3, \dots, \texttt{false})$
$\forall(a_1, \exists\,(a_2, \forall(a_3, \dots, \texttt{false})$
$-\nexists\, a_1$
$-\exists\,(a_1, \nexists\, a_2)$
proper
non-proper

Figure 4.2: Illustration of proper conditions[5]

**Example 34.** The linear constraints

$$\forall(\ \boxed{\text{Pl}}\ , \exists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ )$$

$$\forall(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}}\ , \texttt{false}\ ) \qquad \equiv \nexists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}}$$

$$\exists\,(\ \boxed{\text{Pl}}\ , \forall\ \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ , \texttt{false}) \qquad \equiv \exists\,(\ \boxed{\text{Pl}}\ , \nexists\ \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ )$$

are proper. The linear constraint

$$\forall(\ \boxed{\text{Pl}}\ , \exists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ , \forall\ \boxed{\text{Pl}} \overset{\xrightarrow{\text{tok}}}{\underset{\xrightarrow[\text{tok}]{}}{}} \boxed{\text{Tk}}\ , \texttt{false}) \equiv \forall(\ \boxed{\text{Pl}}\ , \exists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ , \nexists\ \boxed{\text{Pl}} \overset{\xrightarrow{\text{tok}}}{\underset{\xrightarrow[\text{tok}]{}}{}} \boxed{\text{Tk}}\ )$$

is non-proper.

[5] In the figure, the minus symbol means, that the conditions are excluded from this side.

Proper conditions turn out to be satisfiable.

**Lemma 18.** Every proper condition is satisfiable.

**Proof (By induction along the nesting structure of the condition).** By the equivalence rules from Fact 2, proper conditions are satisfiable: For conditions of nesting depth $n = 0$, the proper condition is `true`, which is satisfiable. For conditions of nesting depth $n = 1$, the proper conditions are $\exists\,(x, \texttt{true}) \equiv \exists\, x$, $\forall(x, \texttt{true}) \equiv \texttt{true}$, which are both satisfiable. For the proper condition $\forall(x, \texttt{false}) \equiv \nexists\, x$, $x$ is a real morphism. Consequently, it is satisfiable. For conditions of nesting depth $n \to n + 1$, there are two cases.

(1) Let $d = \forall(x, c)$, then, by Definition 20, $d$ ends with `true`, by induction hypothesis, $c$ is satisfiable, and consequently, $d$ is satisfiable.

(2) Let $d = \exists\,(x, c)$. Since $d$ is a condition with alternating quantifiers, there are two cases: 1. $c$ has a $\forall$-quantifier, by induction hypothesis, $c$ is satisfiable, $d$ ends with `true`, and consequently, $d$ is satisfiable. 2. $c$ has a $\nexists$-quantifier, then, by Definition 20, $d$ is of the form $\exists\,(x, \nexists\, b)$, and $b$ is a real morphism, consequently, it is satisfiable. □

Note that also some non-proper conditions may be satisfiable, e.g., $\forall(\underset{1}{\circ}\ \ \underset{2}{\circ}, \exists(\underset{1}{\circ}\!\longrightarrow\!\underset{2}{\circ}, \forall(\underset{1}{\circ}\rightleftarrows\underset{2}{\circ}, \texttt{false})))$ (see the remark at the end of the section). For proper conditions, repair programs based on the solid (ad hoc) construction can be constructed.

**Theorem 2 (Repair I).** There is a repair program for proper conditions based on the solid (ad hoc) construction.

For the construction of repair programs for proper conditions, it is crucial to hand over information between the transformation steps. To do so, we make use of the statements `Mark` and `Unmark` to restrict the applicability of a program to a marked context. $\texttt{Mark}(a) = \langle a, \mathrm{id}_C\rangle$ is the rule with left interface $a$ and identical plain rule $\mathrm{id}_C = \langle C \hookleftarrow C \hookrightarrow C\rangle$. Given an occurrence of $A$, it is used for a marking of an occurrence of $C$, extending the occurrence of $A$. Similar, $\texttt{Mark}(a, \mathrm{ac}) = \langle a, \mathrm{id}_C, \mathrm{ac}\rangle$ is used for marking an occurrence of $C$ satisfying the condition ac. $\texttt{Unmark}(a) = \langle \mathrm{id}_C, a\rangle$ is the identical plain rule with right interface $a$, used for unmarking the occurrence of $C$.

**Construction 13.** For proper conditions $d$, the solid (ad hoc) repair programs are constructed inductively as follows.

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

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

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

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

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

where $a\colon A \hookrightarrow C$ is real, $\mathcal{R}_a$ and $\mathcal{S}'_a$ are the sets according to the solid (ad hoc) construction, and $P_c$ is a repair program for $c$ with interface $C$.

**Remark.** The constructions for the condition $\exists\,(a, c)$ and $\forall(a, c)$ make use of the existence of a repair program for $c$, not the properness of $c$. In the same way, one can construct a repair program for condition $\exists\,(a, c)$ and $\forall(a, c)$ with non-proper $c$, but existing repair program.

**Informal Description.** The repair programs work as follows. (1) For the condition $\texttt{true}$, there is nothing to do. (2) For positive conditions $\exists\, a$, for an occurrence of $A$, we try to complete it to an occurrence of $C$, via the set $\mathcal{R}_a$. Otherwise, the condition is satisfied and we do nothing. (3) For negative conditions $\nexists\, a$, for each occurrence of $C$, we delete as least as possible of it via the program $\mathcal{S}'_a\!\downarrow$. To do so, we delete edges, whenever possible, and only delete nodes, if the morphism is not edge-increasing. For the deletion of nodes, we have to delete the dangling edges first, then apply the rule as in the double-pushout approach. (4) For existential conditions $\exists\,(a, c)$, we try to repair $\exists\, a$, then we mark one occurrence of $C$, apply the program for $c$, and finally unmark. To do so, we make use of the interfaces. (5) For universal conditions $\forall(a, c)$, we mark each occurrence of $C$ for which the condition $c$ is not satisfied, then we apply the repair program $P_c$ for the condition $c$, and unmark. By the application of the rules as long as possible, we repair each occurrence of $C$. By $\texttt{Mark}(a, \neg c)$, we fix one occurrence of $C$ by making use of the interfaces. If $\texttt{Mark}(a, \neg c)$ cannot be applied any more, each occurrence of $C$ satisfies $c$. This guarantees termination: each occurrence of $C$ can only be marked at most once.

**Example 35.** 1. Given the constraint $d = \exists\,(\,\boxed{\text{Pl}}\,, \nexists\ \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\,)$ meaning there exists a Pl-node which does not possess two Tk-nodes and the connecting containment tok-edges. The constraint is of the form $\exists\,(a, c)$, with $a\colon \emptyset \hookrightarrow \boxed{\text{Pl}}$,

$c = \nexists\, b$, and $b\colon \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Tk}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}$. By Theorem 2(4), the repair program for $d$ is $P_d = \langle \mathtt{try}\ \mathcal{R}_a; \langle \mathtt{Mark}(a); P_c; \mathtt{Unmark}(a)\rangle\rangle$. The condition $c$ is negative, and by Theorem 2(3), $P_c = \mathcal{S}'_b{\downarrow}$, and $\mathcal{S}_b$ is the repairing set from Example 33. The program tries to add a Pl-node. It marks a Pl-node and, if there are two tokens and connecting containment tok-edges, it deletes one edge (see Figure 4.3). The Pl-node is added at most one time, the deletion is done as long as possible.

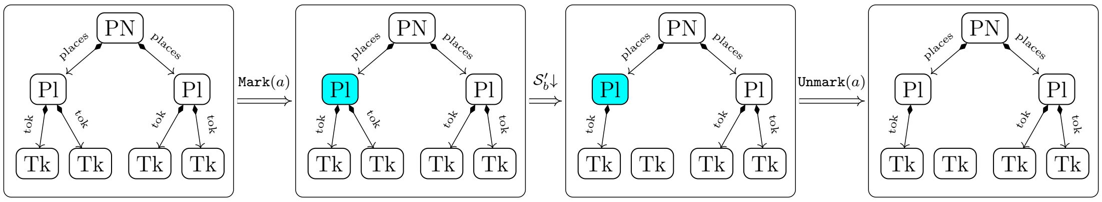


Figure 4.3: Transformation via the repair program

2. Given the constraint $d = \forall(\ \boxed{\mathrm{Pl}}\ , \exists\ \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}\ )$, meaning that for each Pl-node, there exists a Tk-node and the tok-edge, a repair program for $d$ can be constructed according to Theorem 2. The constraint $d$ is of the form $\forall(a, c)$ with morphism $a\colon \emptyset \hookrightarrow \boxed{\mathrm{Pl}}$ and condition $c = \exists\ \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}$ . By Theorem 2(5), the repair program for $d$ is $P_d = \langle \mathtt{Mark}(a, \neg c); P_c; \mathtt{Unmark}(a)\rangle{\downarrow}$. The condition $c$ is of the form $\exists\, b$. By Repair Theorem 2(2), the repair program for $c$ is $P_c = \mathtt{try}\ \mathcal{R}_b$, where $\mathcal{R}_b$ is the rule set $\mathcal{R}_a$ from Example 32. The core program $\langle \mathtt{Mark}(a, \neg c); P_c; \mathtt{Unmark}(a)\rangle$ marks an occurrence of a Pl-node without connecting containment edge to a Tk-node (see Figure 4.4). The program $P_c$ tries to add a Tk-node and the containment edge, provided there do not exist a Pl-node and a Tk-node, and to add a containment edge between a Pl- and Tk-node, provided that there does not exist such an edge to another Tk-node. Finally, the marked part is unmarked. The core program is applied as long as possible. For the left graph in Figure 4.4, after one application of the core program, the constraint $d$ is satisfied.

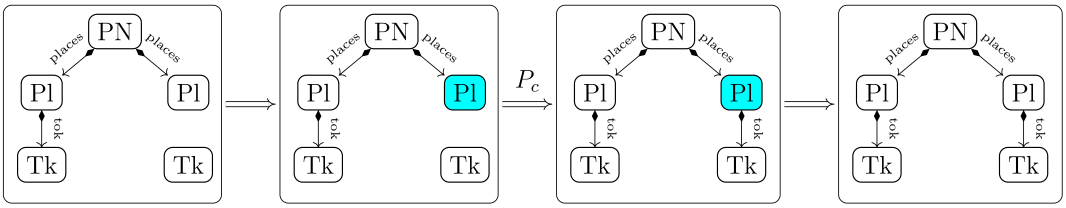


Figure 4.4: Transformation via the repair program

The proof of Theorem 2 makes use of the fact the rules in $\mathcal{R}_a$ are increasing and the rules in $\mathcal{S}_a$ are decreasing.

A rule $\varrho = \langle L \hookleftarrow K \hookrightarrow R, \mathrm{ac}\rangle$ is *decreasing* (*increasing*) if $L \supset K \cong R$ ($L \cong K \subset R$). A program $P$ is *decreasing* (*increasing*) if all rules in $P$ are.

**Fact 7.** For negative conditions, the solid (ad hoc) repair program is decreasing. For positive, existential, and universal conditions, the solid (ad hoc) repair program is increasing.

**Proof (of Theorem 2). By induction on the structure of the condition.**

We show the correctness and existence of the solid repair programs for graphs, the ad hoc repair programs for graphs, and typed graphs.

**1. Solid repair programs.**

**Correctness.** Let $d$ be a proper condition. Let (1) $P_d$ be the program based on the solid construction and (2) $\widehat{P_d}$ be the program based on the ad hoc construction.

(1) Let $d = \mathtt{true}$. For all triples $\langle g, h, i\rangle \in [\![\mathtt{Skip}]\!]$, $h \circ i \models \mathtt{true}$, i.e., $\mathtt{Skip}$ is a repair program for $\mathtt{true}$.

(2) Let $d = \exists\, a$, $P_d = \mathtt{try}\ \mathcal{R}_a$ with $\mathcal{R}_a$ as in Construction 12, and $g \Rightarrow_{\mathtt{try}\ \mathcal{R}_a, i} h$.

(a) If $g \models \nexists\, a$, then there is a decomposition of the morphism $g$ into morphisms $b\colon A \hookrightarrow B$ and $g'\colon B \hookrightarrow G$ such that $g' \models \mathrm{Shift}(b, \nexists\, a)$. Then the rule $\varrho = \langle b, B \Rightarrow C, \mathrm{ac}, a\rangle$ in $\mathcal{R}_a$ is applicable yielding morphisms $h'$ and $h = h' \circ a$ (see below). By construction, $h' \models \mathrm{Shift}(b, \exists\, a)$ and, by the Shift Lemma, $h \circ i \models \exists\, a$.

$$\begin{tikzcd}
& A \arrow[dl, hook', "b"'] \arrow[dr, hook, "a"] \arrow[ddl, bend right=40, hook', "g"'] \arrow[ddr, bend left=40, hook, "h"] & \\
B \arrow[rr, Rightarrow] \arrow[d, hook, "g'"'] & & C \arrow[d, hook, "h'"] \\
G \arrow[rr, Rightarrow] & & H
\arrow[from=2-1, to=3-3, phantom, "="]
\end{tikzcd}$$

(b) If $g \models \exists\, a$, then, by the semantics of $\mathtt{try}$, $g \Rightarrow_{\mathcal{R}_a, i} g = h \circ i \models \exists\, a$.

(3) Let $d = \nexists\, a$ and $g \Rightarrow_{\mathcal{S}'_a\downarrow, i} h$. By the semantics of $\downarrow$, the program $\mathcal{S}'_a$ is not applicable to $h$. Then, for every injective morphism $h\colon A \hookrightarrow H$ and every rule $\langle a, C \Rightarrow B, b\rangle$ in $\mathcal{S}_a$, there is no injective morphism $h'\colon C \hookrightarrow H$, such that $h' \circ a = h$, i.e., $h \models \nexists\, a$ (see Figure below).

$$\begin{tikzcd}
C \arrow[dr, hook, "h'"'] & & A \arrow[ll, hook', "a"'] \arrow[dl, hook', "h", "/" marking] \arrow[r, phantom, "\vartriangleleft"] \arrow[dr, phantom, "\not\models"] & \exists\, a \\
& H \arrow[u, phantom, "="] & &
\end{tikzcd}$$

(4) Let $d = \exists(a, c)$ and $g \Rightarrow_{P_d} h$. By Definition of $P_d$ the transformation is of the form $g \underset{P_{\exists a}, i_1}{\Longrightarrow} h_1 \underset{\texttt{Mark}(a), a}{\Longrightarrow} h_2 \underset{P(c), i_2}{\Longrightarrow} h_2 \underset{\texttt{Unmark}(a), a^{-1}}{\Longrightarrow} h$ where $i_1, i_2$ are identities (see Figure below).

$$\begin{tikzcd}
 & A \arrow[r, "i_1"] \arrow[dl, "g"'] & A \arrow[rr, hook, "a"] \arrow[dr, "h_1"] & = & C \arrow[r, "i_2"] \arrow[dl, "h_2"'] & C \arrow[dr, "h_3"] & = & A \arrow[ll, hook', "a"'] \arrow[dl, "h"'] \\
G \arrow[rrr, Rightarrow, "P_{\exists a}"'] & & & H_1 \arrow[rrr, Rightarrow, "P_c"'] & & & H &
\end{tikzcd}$$

By induction hypothesis, $P_{\exists a}$ is a repair program for $\exists a$ and $P_c$ is a repair program for $c$, i.e., $h_1 \circ i_1 \models \exists a$ and $h_3 \circ i_2 \models c$. By the semantics of $\texttt{Mark}$ and $\texttt{Unmark}$, $h_1 = h_2 \circ a$ and $h = h_3 \circ a$. Consequently, there is some injective morphism $h_3 \colon C \hookrightarrow H$ with $h = h_3 \circ a$ and $h_3 \models c$, i.e., $h \circ i \models \exists(a, c)$ where $i$ is the identity.

(5) Let $d = \forall(a, c)$ and $g \Rightarrow_{P_d} h$. Let $P_d = P'_d{\downarrow}$ with subprogram $P'_d = \langle \texttt{Mark}(a, \neg c); P_c; \texttt{Unmark}(a) \rangle$. By Definition of $P_d$, the transformation is of the form $g \Rightarrow_{\texttt{Mark}} h_0 \Rightarrow_{P_c} h_1 \Rightarrow_{\texttt{Unmark}} h$.

$$\begin{tikzcd}
A \arrow[rr, hook, "a"] \arrow[dr, "h"'] & = & C \arrow[dl, "h'", "/" marking] & \lhd \ \neg c \\
 & H & \not\models &
\end{tikzcd}$$

By induction hypothesis, $P_c$ is a repair program for $c$. By the semantics of $\downarrow$, the program $P'_d$ is not applicable to $h$, i.e., $\texttt{Mark}(a, \neg c)$ is not applicable, and there is no morphism $h'$ such that $h = h' \circ a$ and $h' \models \neg c$, i.e., $h \models \neg\exists(a, \neg c) \equiv \forall(a, c)$.

This completes the inductive proof for graphs.

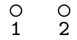
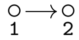
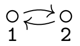
**Existence.** By assumption, the condition $d$ over $A$ is proper. By Lemma 18, $d$ is satisfiable, thus, there is a morphism $g \colon A \hookrightarrow G$ the satisfies $d$. By Construction, $P_d$ is a program with interfaces $A$ and the domain of $g$ and $i$ is $A$. Consequently, the program is applicable to every $g$ yielding a transformation $g \Rightarrow h$, i.e., $\forall g \exists\, g \Rightarrow_{P_d, i} h$.

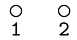
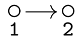
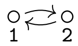
**2. Ad hoc repair programs.** Replacing $\mathcal{R}_a$ and $\mathcal{S}_a$ by $\widehat{\mathcal{R}_a}$ and $\widehat{\mathcal{S}_a}$, respectively, we obtain the proof for the ad hoc repair programs. For positive conditions, if $g \models \nexists a$, then the rule $\langle b, B \Rightarrow C', \mathrm{ac}, a' \rangle$ in $\widehat{\mathcal{R}_a}$ is applicable, $\langle g, h, i \rangle \in \llbracket \texttt{try}\ \widehat{\mathcal{R}_a} \rrbracket$ and $h \circ i \models \exists a$. For negative conditions let $\langle g, h, i \rangle \in \llbracket \widehat{\mathcal{S}_a}' {\downarrow} \rrbracket$, by the semantics of $\downarrow$, $\widehat{\mathcal{S}_a}'$ is not applicable to $h \circ i$, i.e., $h \circ i \models \nexists a$.

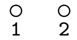
**3. Typed graphs**: For every morphism $a \colon A \hookrightarrow C$, and every proper subgraph $B$ of $C$, define $\mathit{type}_B = \mathit{type}_C \circ \mathrm{inc}_B$, where for $B \subseteq C$, $\mathrm{inc}_B$ denotes the inclusion of $B$ in $C$. Then $\mathit{type}_A = \mathit{type}_B \circ \mathrm{inc}_A$ and the rules $\langle b, B \Rightarrow C, a \rangle \in \mathcal{R}_a$ and $\langle a, C \Rightarrow B, b \rangle \in \mathcal{S}_a$ consist of typed graph morphisms. In this way the graphs in the rules in $\mathcal{R}_a$ and $\mathcal{S}_a$ become typed. The typing of the conditions is a direct consequence. □

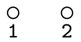
**Remark.** Theorem 2 can be formulated for a larger class of conditions:

1. Beside proper conditions, for all conditions equivalent to a proper condition, there is a repair program. The condition $d$ below is equivalent to a proper one $d'$:

$d = \forall($  $, \exists($  $, \forall($  $, \texttt{false})))$

$\equiv \forall($  $, \exists($  $, \nexists($  $)))$

$\equiv \forall($  $, \texttt{false})$

$\equiv \nexists$  $= d'$.

2. The construction of a repair program for a condition of the form $\exists(a, c)$ (or $\forall(a, c)$) can be done, provided that there exists a repair program for $c$. Properness only guarantees the existence of a repair program[6].

## 4.4 Conjunctive and disjunctive conditions

In the following, we consider conjunctions and disjunctive of conditions with repair programs. A condition is a *condition with repair program* if there is a repair program for the condition, i.e., conditions for which a repair program exists.

### Conjunctive conditions

For conjunctive conditions, our aim is to construct a repair program from the repair programs of the conditions in the conjunction with the divide and conquer method:

**Idea.** Given a conjunction of conditions and the corresponding repair programs.

(1) Decompose the conjunction into subconditions.

(2) Try to find a so-called preserving sequentialization of the subconditions.

(3) Compose the repair programs for the subconditions to a sequential repair program for the conjunctive condition.

An essential topic is the preservation of conditions: Let $P_1$ and $P_2$ be repair programs for $d_1$ and $d_2$, respectively. If the repair program $P_2$ does not preserve the condition $d_1$, the application of $P_2$ to a graph, which has previously been repaired by $P_1$, may yield a graph that does not satisfy $d_1 \wedge d_2$. To solve this problem, we consider condition-preserving programs and condition-preserving repair programs.

[6] We have focused on the syntactical construction of repair program for proper conditions.

In Habel and Pennemann [HP09], the preservation of conditions by a rule is considered. We generalize this definition to the preservation of conditions by programs.

**Definition 21 (preservation).** A program $P$ is *$d$-preserving* if every rule in $P$ is $d$-preserving.

By [HP09, Corollary 5], for every rule $\varrho$ and every condition $d$, an application condition $\mathrm{Pres}(\varrho, d)$ can be constructed, such that the rule together with the application condition is $d$-preserving (see Section 2.4). This result can be generalized to conditions and programs: the replacement of every rule in the program by the corresponding $d$-preserving rule yields a $d$-preserving program.

**Lemma 19 (preservation).** For every program $P$ and every condition $d$, there is a $d$-preserving program $P^d$.

**Construction 14.** The program $P^d$ is constructed from the program $P$ by replacing all rules $\varrho$ in $P$ by the condition-preserving rule. In more detail, the program is constructed inductively: For rules $\varrho$, the $d$-preserving rule is $\langle \varrho, \mathrm{Pres}(\varrho, d)\rangle$, where $\mathrm{Pres}(\varrho, d)$ is according to Construction 3. For programs with interface, the $d$-preserving programs are obtained by integrating the condition $d$ into each subprogram, i.e., $\{P, Q\}^d = \{P^d, Q^d\}$, $\langle P; Q\rangle^d = \langle P^d; Q^d\rangle$, $P{\downarrow}^d = P^d{\downarrow}$, and $(\texttt{try}\ P)^d = \texttt{try}\ P^d$.

**Proof.** By Construction 3, all rules in $P^d$ are $d$-preserving, thus, the program is $d$-preserving. □

If the program $P$ is already $d$-preserving, the $d$-preserving program $P^d$ is equivalent to the program $P$.

**Fact 8.** $P^d \equiv P$ provided that $P$ is $d$-preserving.

**Remark.** In special cases, the application conditions in the program may be so restrictive, such that the program cannot be applied to any graph, i.e., $P^d \equiv \bot$, and the program does nothing.

**Example 36.** Consider the constraint $d = \nexists(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Tk}}\ )$ and the program $P = \langle \texttt{Mark}(a, \neg c); \texttt{try}\ \mathcal{R}_b; \texttt{Unmark}(a)\rangle{\downarrow}$ with $a\colon \emptyset \hookrightarrow \boxed{\text{Pl}}$, $c = \exists(\ \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ )$, and $\mathcal{R}_b$ the rule set from Example 32. By Construction 14, the $d$-preserving program $P^d$ is constructed by replacing every rule $\varrho_i \in \mathcal{R}_b$ with the rule $\varrho_i'$, where $\varrho_i' = \langle \varrho_i, \mathrm{ac}_i \rangle$, and $\mathrm{ac}_i$ is the application condition $\mathrm{ac}_i = \mathrm{Pres}(\varrho_i, d)$ (see Example 11), i.e.,

$$\mathrm{ac}_1 = \mathrm{Pres}(\varrho_1, d) = \nexists\,( \underset{1}{\boxed{\mathrm{Pl}}} \xrightarrow{\mathrm{tok}} \underset{2}{\boxed{\mathrm{Tk}}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} ) \wedge \ldots \;\Rightarrow \nexists\,( \underset{1}{\boxed{\mathrm{Pl}}} \xrightarrow{\mathrm{tok}} \underset{2}{\boxed{\mathrm{Tk}}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} ) \wedge \ldots$$

$$\mathrm{ac}_2 = \mathrm{Pres}(\varrho_2, d) = \nexists\,( \underset{1}{\boxed{\mathrm{Pl}}} \xrightarrow{\mathrm{tok}} \underset{2}{\boxed{\mathrm{Tk}}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} ) \wedge \ldots \;\Rightarrow \nexists\,( \underset{1}{\boxed{\mathrm{Pl}}} \quad \underset{2}{\boxed{\mathrm{Tk}}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} ) \wedge \ldots .$$

By the $d$-preserving application condition, the program is not applicable, provided it introduces a new violation of $d$. This may be seen at the transformation in Figure 4.5: the right-most Pl-node cannot be connected with the Tk-node because this is forbidden by the application condition $\mathrm{ac}_2$.

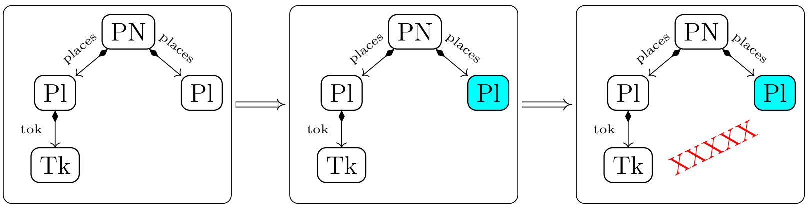


Figure 4.5: The condition-preserving program may be not a repair program

**Remark (ad hoc construction).** In Example 36, the program uses the present Tk-node to add the required tok-edge. The ad hoc repair program is based on the repairing set

$$\widehat{\mathcal{R}_b} \supseteq \left\{ \begin{array}{lll} \langle\, b_1, \boxed{\mathrm{Pl}} & \Rightarrow & \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}\,, \nexists\, \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}\,, a \,\rangle \\ \langle\, b_2, \boxed{\mathrm{Pl}} \;\; \boxed{\mathrm{Tk}} & \Rightarrow & \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}\,, \nexists\, \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}} \wedge \nexists\, \boxed{\mathrm{Tk}} \;\; \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}\,, a \,\rangle \end{array} \right.$$

which allows to add a new Tk-node and the required tok-edge (where the left interface morphisms are $b_1\colon \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}}$ and $b_2\colon \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}} \; \boxed{\mathrm{Tk}}$ and the right interface morphisms are the morphism $a$ of the condition).

The difference to $\mathcal{R}_b$ from Example 32 is the application condition of the first rule: by the ad hoc construction, the application condition of the first rule is $\mathrm{ac} = \mathrm{Shift}(b_1, \nexists\, a) = \nexists\, \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}$ . This makes the first rule applicable at the Pl-node in consideration, and adds a Tk-node together with a tok-edge (see Figure 4.6). In general, this construction may not terminate. The solid repair program is designed to be terminating; the ad hoc repair program might be not terminating.

Beside of conjunctions of conditions (with repair programs) we consider sequences of conditions.

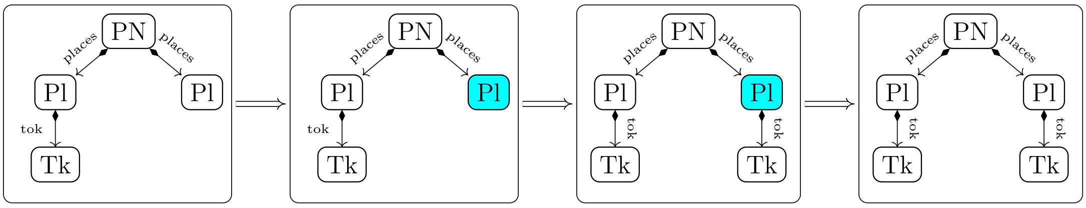


Figure 4.6: The condition-preserving ad hoc program may be used for repair

**Convention.** In the following, let $ds = d_1, \ldots, d_n$ and $Ps = P_1, \ldots, P_n$ be a sequence of conditions and programs, respectively, $P_i$ be a repair program for $d_i$, and $ds_1 = d_1, \ldots, d_k$, $ds_2 = d_{k+1}, \ldots, d_n$ be sequences of conditions with conjunctions $e_1 = \wedge_{i=1}^{k} d_i$ and $e_2 = \wedge_{i=k+1}^{n} d_i$.

A sequence of programs is preserving if for each natural number $k$, the respective repair program $P_k$ preserves all preceding conditions, i.e., $P_1$ is a repair program for $d_1$, $P_2$ is a repair program for $d_2$ and $d_1$-preserving, $P_3$ is a repair program for $d_3$ and $d_1 \wedge d_2$-preserving, and so on (see Figure 4.7).

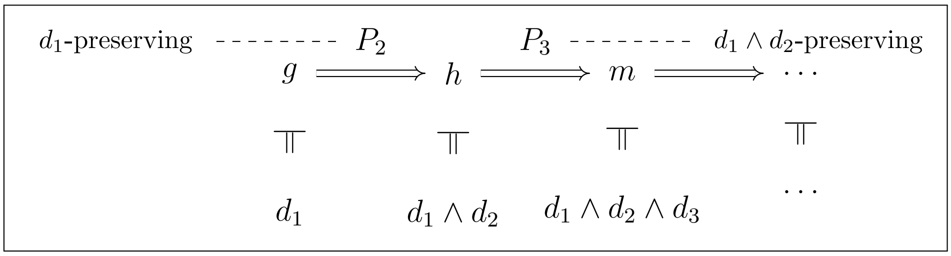
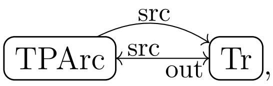
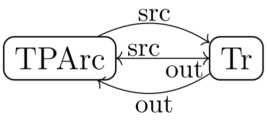


Figure 4.7: Preservation of sequences of conditions

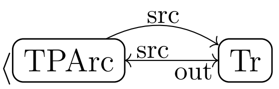
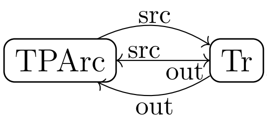
**Definition 22 (preservation).** A sequence $Ps$ is *ds-preserving* (and $ds$ is *preserving*) if, for $k = 2, \ldots, n$, $P_k$ is $\wedge_{i=1}^{k-1} d_i$-preserving. A conjunction is *preserving* if there is a preserving sequentialization.

**Example 37.** Consider the constraints

$$d_1 \quad = \quad \forall(\ \boxed{\text{Pl}}\ , \exists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ ) \text{ and}$$

$d_2 \quad = \quad \forall(a, c) = \forall($  $, \exists$  $)$

meaning that for all Pl-nodes, there exists a Tk-node, and, for all src-edges from a TPArc-node to a Tr-node in the opposite edge relation, there exists an out-edge in opposite direction from a Tr-node to a TPArc-node. The repair program $P_1$ for $d_1$ is given in Example 35. The repair program $P_2$ for $d_2$ is

$P_2 = \langle \texttt{Mark}(a, \neg c); \langle$  $\Rightarrow$  $, \neg c\rangle; \texttt{Unmark}(a)\rangle\downarrow.$

The sequence $P_1, P_2$ is $d_1, d_2$-preserving, after the application of $P_1$, each Pl-node possesses a Tk-node. Afterwards, for each the src-edges between a TPArc-node and a Tr-node, an out-edge in opposite direction is added, while all Tk-nodes are preserved.

The construction of repair programs relies on finding a preserving sequentialization of a given condition (see Figure 4.8).Unfortunately, this does not work for arbitrary sequences of conditions (see Fact 10 and Fact 11).

**Requirement.** A sequence of conditions has to be preserving.

$$\begin{tikzcd}
\text{conjunction } d \arrow[r, "\text{sequentialization}"] \arrow[d, dash, dashed] & d_1, \ldots, d_n \quad \text{preserving} \arrow[d, shift right=4ex] \arrow[d, shift left=2ex] \\
\text{conjunction } d & \langle P_1; \ldots ; P_n\rangle \arrow[l, "\text{repair}"']
\end{tikzcd}$$

Figure 4.8: Sequentialization of conjunctive conditions

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

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

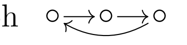
**Fact 9.** If $ds$ is negative (or positive), then $ds$ is $ds$-preserving.

In general, for a repair program of a conjunction, the programs for the conditions in the conjunction cannot be applied in arbitrary order and sometimes cannot be ordered.

**Fact 10.** In general, not every sequentialization $ds$ of conditions is $ds$-preserving.

**Example 38 (not every sequentialization is preserving).** Consider, e.g., the constraints $d_1 = \forall(\ \circ\ , \exists\ \circ\!\circlearrowright\ )$ and $d_2 = \forall(\ \circ\ , \exists\ \circ\!\rightarrow\!\circ\ )$ with the repair programs $P_1$ and $P_2$ constructed according to Construction 13. For the sequentialization $d_1, d_2$, the program $P_2$ does not preserve the constraint $d_1$: For a node with loop satisfying $d_1$ the program $P_2$ adds a new node and a connecting edge. The new node does not have a loop, i.e., the resulting graph does not satisfy $d_1$. For the sequentialization $d_2, d_1$, the program $P_1$ preserves the constraint $d_2$.

**Fact 11.** There are conditions without a preserving sequentialization.

**Example 39 (no preserving sequentialization).** Consider, e.g., the constraints $d_1 = \forall(\ \circ\ , \exists\ \circ\!\rightarrow\!\circ\ )$ and $d_2 = \forall(\ \circ\!\rightarrow\!\circ\ , \exists\ \circ\!\rightarrow\!\circ\quad\circ\ )$ with the repair programs $P_1$ and $P_2$ constructed according to the solid construction. The condition $d_1 \wedge d_2$ is satisfiable: the graph  satisfies $d_1 \wedge d_2$. The program $P_2$ does not preserve $d_1$ and $P_1$ does not preserve $d_2$: Application of $P_2$ to $\circ\!\rightleftarrows\!\circ\ \models d_1$ yields to $\circ\!\rightleftarrows\!\circ\quad\circ\ \not\models d_1$ and application of $P_1$ to $\circ\ \models d_2$ yields to $\circ\!\rightleftarrows\!\circ\ \not\models d_2$.

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

**Lemma 20 (preserving repair).** If $d_1, \ldots, d_n$ is preserving, then $\langle P_1; \ldots; P_n\rangle$ is a repair program for $\wedge_{i=1}^{n} d_i$.

**Proof.** By induction on the number $n$ of conditions with the repair programs. For $n$=1, by Theorem 2, $P_1$ is a repair program for $d_1$. Inductive hypothesis: If $P_2, \ldots, P_n$ is $d_1, \ldots, d_n$-preserving, then $P = \langle P_1; \ldots; P_n\rangle$ is a repair program for the conjunction $\wedge_{i=1}^{n} d_i$. Inductive step: For $n \rightarrow n+1$, let $P_2, \ldots, P_{n+1}$ be $d_1, \ldots, d_{n+1}$-preserving. Then $P_2, \ldots, P_n$ is $d_1, \ldots d_n$-preserving and, by induction hypothesis, the program $P = \langle P_1; \ldots; P_n\rangle$ is a repair program for the conjunction $d = \wedge_{i=1}^{n} d_i$. Moreover, $P_{n+1}$ is a $d$-preserving repair program for $d_{n+1}$. Consequently, for every transformation $g \Rightarrow_P g_n \Rightarrow_{P_{n+1}} h$, $g_n \models d$ and $h \models \wedge_{i=1}^{n+1} d_i$. Thus, $\langle P_1; \ldots; P_{n+1}\rangle$ is a repair program for $\wedge_{i=1}^{n+1} d_i$. □

**Example 40.** We continue with Example 37. The repair program $P_2$ for $d_2$ is $d_1$-preserving. Consequently, $\langle P_1; P_2\rangle$ is a repair program for $d_1 \wedge d_2$.

**Assumption 3.** In the following, we try to construct solid repair programs (if not stated otherwise). The presented constructions (Lemma 21, Theorem 3) also hold for ad hoc repair programs. The reason for this is that all proofs rely on the fact, that the repair programs are increasing (or decreasing).

In the following, we consider a conjunction of negative and universal conditions and try to construct solid repair programs. Let $e_1$ be the conjunction of negative and $e_2$ the conjunction of universal conditions. By Fact 9, every sequence of negative conditions is preserving and, by Lemma 20, the sequential composition $Q_1 = \langle P_1; \ldots; P_k\rangle$ ($k \in \mathbb{N}$) of the repair programs forms a repair program for the conjunction of negative conditions $e_1$. In general, not every sequentialization of universal conditions is preserving. We have to require preservation. In the case of preservation, the sequential composition $Q_2 = \langle P_{k+1}; \ldots; P_n\rangle$ of the repair programs forms a repair program for the conjunction of universal conditions $e_2$. But the repair program $Q_2$ may be not $e_1$-preserving (see Figure 4.9). By Lemma 19, we can replace each rule $\varrho$ in $Q_2$ by the corresponding $e_1$-preserving ones. The resulting program $Q_2^{e_1}$ may be no repair program for $e_2$.

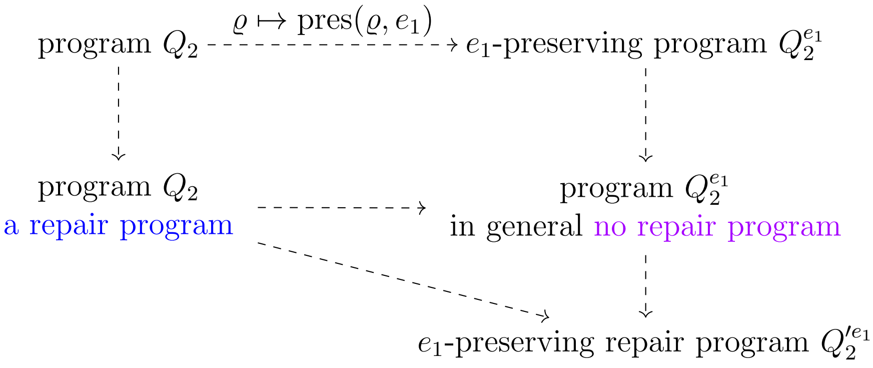

$\varrho \mapsto \mathrm{pres}(\varrho, e_1)$
program $Q_2$
$e_1$-preserving program $Q_2^{e_1}$
program $Q_2$
a repair program
program $Q_2^{e_1}$
in general no repair program
$e_1$-preserving repair program $Q_2'^{e_1}$

Figure 4.9: Construction of an $e_1$-preserving repair program

By Lemma 21 below, $Q_2$ can be modified to an $e_1$-preserving repair program $Q_2'^{e_1}$ for $e_2$. The idea is to delete all occurrences of the morphism $a$ of the universal condition $\forall(a, c)$, which violate the condition $c$. Given a universal condition, we mark an occurrence of the morphism violating the condition, then the occurrence of the morphism at that position is deleted. To mark the morphism at that position, we modify the left interface to the identity of the codomain of the morphism. For a conjunction of universal conditions, an $e_1$-preserving repair program can be constructed from the $e_1$-preserving program by destroying all non-repaired occurrences of all the universal conditions. By Fact 13 below, the sequential composition $\langle Q_1; \mathrm{Q}_2'^{e_1}\rangle$ becomes a repair program for the conjunction $e_1 \wedge e_2$.

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

**Lemma 21 (preserving repair program).** If $e_1$ is a negative conjunction and $Q_2$ is a repair program for a preserving universal conjunction $e_2$, then there is an $e_1$-preserving repair program $Q_2'^{e_1}$ for $e_2$.

**Construction 15.** For a conjunction $e_1$ of negative conditions and a single universal condition $e_2 = \forall(a, c)$ with repair program $P$, let

$$P'^{e_1} = \langle P^{e_1}; P^{\mathrm{id}}_{\nexists a}\rangle$$

where $P^{\mathrm{id}}_{\nexists a} = \langle \mathtt{Mark}(a, \neg c); \mathcal{S}'^{\mathrm{id}}_a\rangle\downarrow$ where $\mathcal{S}'^{\mathrm{id}}_a$ is obtained from $\mathcal{S}'_a$ by replacing the left interface morphism $a\colon A \hookrightarrow C$ by the identity $\mathrm{id}\colon C \hookrightarrow C$. For a preserving conjunction $e_2$ of universal conditions with sequentialization $d_1, \ldots, d_n$ and repair program $Q_2 = \langle P_1; \ldots; P_n\rangle$, let

$$Q_2'^{e_1} = \langle P_1'^{e_1}; \ldots; P_n'^{e_1}\rangle.$$

**Example 41.** We continue with Example 36. Consider the constraints $e_1 = \nexists(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}}\ )$ and $e_2 = \forall(\ \boxed{\text{Pl}}\ , \exists(\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ ))$. By Theorem 2, there are repair programs

$$P_1 = \langle\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}}\ \Rightarrow \boxed{\text{Pl}} \quad \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}}\ \rangle\downarrow$$

and $P_2$, which is the program $P$ from Example 36. Since $P_2$ is not $e_1$-preserving, we construct the $e_1$-preserving application conditions, given in Example 36. The $e_1$-preserving program for $e_2$ is not a repair program for $e_2$. Recall the transformation in Figure 4.5: by the application condition, the right-most Pl-node cannot be connected with the Tk-node. Consequently, $e_2$ is not satisfied, and the program $P_2^{e_1}$ is not a repair program. In this situation we can add or delete information to get a repair program. Addition would yield to a program that is not minimally adding. We decide to delete information.

By Lemma 21, there is an $e_1$-preserving repair program $Q_2'^{e_1} = \langle P_2^{e_1}; P^{\mathrm{id}}_{\nexists a}\rangle$ for $e_2$ where $P^{\mathrm{id}}_{\nexists a}$ is a slightly modified version of the repair program $P_{\nexists a}$ for the condition $\nexists a$. In more detail, the program looks as follows:

$$P^{\mathrm{id}}_{\nexists a} = \langle \mathtt{Mark}(\ \boxed{\text{Pl}}\ , \nexists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ ); \langle \mathrm{id},\ \boxed{\text{Pl}}\ \Rightarrow \emptyset\rangle'\rangle\downarrow$$

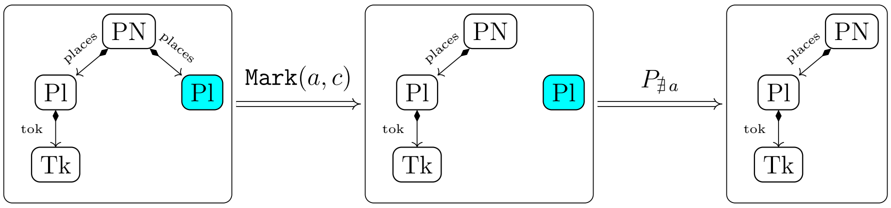


Figure 4.10: Transformation by the $e_1$-preserving repair program

where id is the identity id: $\boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}$. The program marks an occurrence of a Pl-node without incoming containment edge from a Tk-node and deletes (in SPO-style) the occurrence of the Pl-node; this is done as long as possible. This may be seen at the transformation in Figure 4.10.

The rightmost Pl-node does not possess a Tk-node and the connecting containment tok-edge. By the program $P^{\text{id}}_{\nexists a}$ the Pl-node is marked, the dangling edges are deleted, and the Pl-node is deleted. Afterwards, all Pl-nodes in the graph possess a Tk-node and the connecting containment tok-edge, i.e., the graph satisfies the conjunction $e_1 \wedge e_2$ and we obtain a repair program for $e_1 \wedge e_2$. The program deletes a Pl-node, and thus, does not preserve all informations of the modelled Petri net.

**Remark (ad hoc repair).** In Lemma 21, we have constructed preserving solid repair programs. Instead of the solid repair program, we could use an ad hoc repair program, which adds another Tk-node and the tok-edge instead of deleting the Pl-node and the edge. The reason we have chosen a decreasing program for the repairing step in Lemma 21 is (1) that there exist conditions for which the deletion is necessary to satisfy the condition (see Example 47), and (2) that, in general, the ad hoc repair program is not terminating. In practice, the increasing program is often preferred as it does not delete information.

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

2. Let $d_1, \ldots, d_n$ be a sequence of conditions. By assumption, $d_1, \ldots, d_n$ is preserving. Moreover, for $i = 1, \ldots, n$, $P_i$ is a repair program for $d_i$ and, by Lemma 21.1, $P'^{e_1}_i$ is an $e_1$-preserving repair program for $d_i$. Since $d_1, \ldots, d_n$ is preserving, for $k = 2, \ldots, n$, $P'^{e_1}_k$ is $\wedge_{i=1}^{k-1} d_i$-preserving, and, by Lemma 20, $Q'^{e_1}_2 = \langle P'^{e_1}_1; \ldots; P'^{e_1}_n \rangle$

is a repair program for $\wedge_{i=1}^{n} d_i = e_2$. Since all programs in the sequential composition are $e_1$-preserving, the program $Q_2'^{e_1}$ is $e_1$-preserving. Consequently, every transformation $g \Rrightarrow_Q m$ is of the form $g \Rrightarrow_{Q_1} h \Rrightarrow_{Q_2'^{e_2}} m$. Since $Q_1$ is a repair program for $e_1$, $h \models e_1$. Since $Q_2'^{e_2}$ is the $e_1$-preserving repair program for $e_2$, $m \models e_1 \wedge e_2$. Thus, $\langle Q_1; Q_2'^{e_1} \rangle$ is a repair program for $e_1 \wedge e_2$. □

Note that the helper program $P^{\mathrm{id}}_{\nexists a}$ in the $e_1$-preserving repair program is decreasing. As a consequence, the $e_1$-preserving repair program may be not increasing.

**Fact 12.** In general, for conjunctions of negative conditions $e_1$ and preserving conjunctions $e_2$, the $e_1$-preserving repair program $Q_2$ for $e_2$ is not increasing.

Given two conditions with repair programs, then, the first program and the condition-preserving repair program of the second program can be sequentially composed to a repair program for the whole conjunction.

**Fact 13 (composition).** For arbitrary conditions $e_1, e_2$, the following holds. If $Q_1$ is a repair program for $e_1$ and $Q_2'^{e_1}$ is an $e_1$-preserving repair program for $e_2$, then $\langle Q_1; Q_2'^{e_1} \rangle$ is a repair program for $e_1 \wedge e_2$.

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

**Theorem 3 (Repair II).** There is a solid (ad hoc) repair program $P$ for a conjunction $d$ of conditions provided that $d$ is satisfiable, there are solid (ad hoc) repair programs $P_1, \ldots, P_n$ for $d_1, \ldots, d_n$, respectively, there is a sequentialization $ds = d_1, \ldots, d_n$, and

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

where the composition of the sequences $ds_1$ and $ds_2$ yields the sequence $ds$.

**Construction 16.**

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

where $P_1, \dots, P_n$ are repair programs for $d_1, \dots, d_n$, respectively, $ds_1 = d_1, \dots, d_k$, $ds_2 = d_{k+1}, \dots, d_n$ be sequences of conditions, $Q_1{=}\langle P_1; \dots; P_k\rangle$, $Q_2 = \langle P_{k+1}; \dots; P_n\rangle$ are the repair programs for $e_1{=}\wedge_{i=1}^{k} d_i$ and $e_2 = \wedge_{i=k+1}^{n} d_i$, respectively, and $Q_2'^{e_1} = \langle P_{k+1}'^{e_1}; \dots; P_n'^{e_1}\rangle$ is the $e_1$-preserving repair program for $e_2$.

**Proof.** 1. Let $d_1, \dots, d_n$ be negative (positive). Then the repair programs $P_1, \dots, P_n$ are decreasing (increasing) and $d_1, \dots, d_n$ is preserving. By Lemma 20, $\langle P_1; \dots; P_n\rangle$ is a repair program for $\wedge_{i=1}^{n} d_i$.

2. Let $d_1, \dots, d_k$ be positive and $d_{k+1}, \dots, d_n$ universal (or existential) and preserving. By Theorem 3.1 there are repair programs $Q_1 = \langle P_1; \dots; P_k\rangle$ and $Q_2 = \langle P_{k+1}; \dots; P_n\rangle$ for $e_1 = \wedge_{i=1}^{k} d_i$ and $e_2 = \wedge_{i=k+1}^{n} d_i$, respectively. Since $Q_2$ is increasing, it is $e_1$-preserving. By Lemma 20, $\langle Q_1; Q_2\rangle$ is a repair program for $e_1 \wedge e_2 = \bigwedge_{i=1}^{n} d_i$.

3. Let $d_1, \dots, d_k$ be negative and $d_{k+1}, \dots, d_n$ universal and preserving. By Theorem 3.1 there are repair programs $Q_1 = \langle P_1; \dots; P_k\rangle$ and $Q_2 = \langle P_{k+1}; \dots; P_n\rangle$ for $e_1 = \wedge_{i=1}^{k} d_i$ and $e_2 = \wedge_{i=k+1}^{n} d_i$, respectively. By Lemma 21, $Q_2'^{e_1}$ is the $e_1$-preserving repair program for $e_2$. By Fact 13, $\langle Q_1; Q_2'^{e_1}\rangle$ is a repair program for $e_1 \wedge e_2 = \bigwedge_{i=1}^{n} d_i$. An illustration of this part of the proof is given in Figure 4.11.□

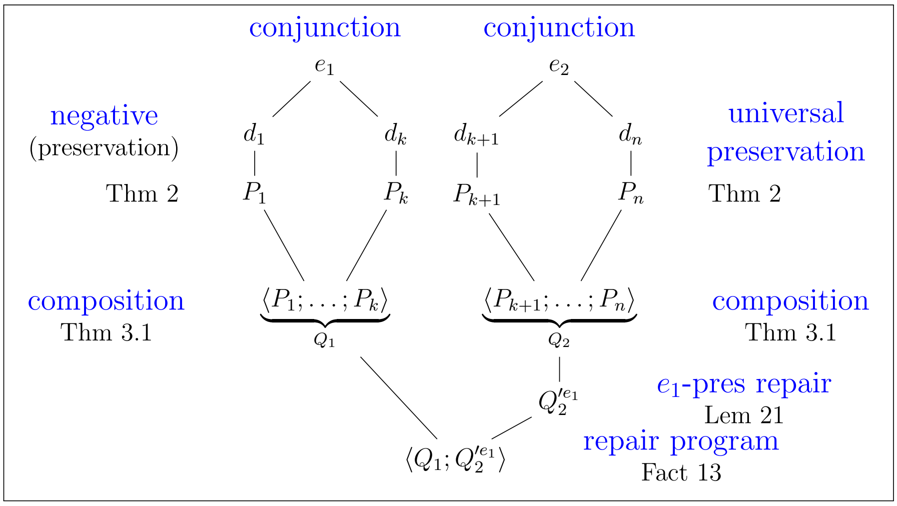


Figure 4.11: Illustration of the proof of Theorem 3.3

## Disjunctive conditions

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

**Theorem 4 (Repair III).**

1. If $P$ is a solid (ad hoc) repair program for $d$ and $d \Rightarrow d'$, then $P$ is a solid (ad hoc) repair program for $d'$.

2. Every solid (ad hoc) repair program for $d_i$ is solid (ad hoc) repair program for $\bigvee_{i=1}^{n} d_i$.

3. If $P_1, \ldots, P_n$ are solid (ad hoc) repair programs for $d_1, \ldots, d_n$, then $\{P_1, \ldots, P_n\}$ is a solid (ad hoc) repair program for $\bigvee_{i=1}^{n} d_i$.

**Example 42.** Consider the constraint $d = d_1 \vee d_2 = \exists\, \boxed{\text{Pl}} \vee \nexists\, \boxed{\text{Pl}} \overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}}$ meaning there exists a Pl-node or there do not exists two parallel containment tok-edges. By Theorem 2, there are repair programs $P_1 = \texttt{try}\ \langle \emptyset \Rightarrow \boxed{\text{Pl}}, \nexists\, \boxed{\text{Pl}} \rangle$ and $P_2 = \langle\ \boxed{\text{Pl}} \overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ \rangle\!\downarrow$. By Theorem 4, the program $P_d = \{P_1, P_2\}$ is a repair program for $d$. It either adds a Pl-node, provided that there does not exist one or it deletes one of the parallel tok-edges, as long as possible.

**Proof.** 1. If $P$ is a repair program for $d$ and $d \Rightarrow d'$, then for every transformation $g \Rightarrow_P h$, $h \models d \Rightarrow d'$, i.e., $P$ is a repair program for $d'$.

2. By $d_i \Rightarrow \bigvee_{i=1}^{n} d_i$ and statement 1, $P_i$ is a repair program for $\bigvee_{i=1}^{n} d_i$.

3. Since $d_i \Rightarrow \bigvee_{i=1}^{n} d_i$ and by statement 1 and 2, $\{P_1, \ldots, P_n\}$ is a repair program for $\bigvee_{i=1}^{n} d_i$ □

## 4.5 Legit conditions

There are repair programs for a large class of conditions. By Theorem 2, proper conditions possess a repair program. Similarly, we may obtain a repair program for conditions that are obtained from a proper one by replacing a subcondition with an arbitrary condition possessing a repair program. In this context, a condition is said to be *generalized proper*. The reason we consider this, is that we can construct solid (ad hoc) repair programs for conditions of the form $\exists\,(a, c)$ (or $\forall(a, c)$), provided that there exists a repair program for $c$. Properness guarantees the existence of a repair program.

The class of legit conditions are all proper or generalized proper ones, preserving conjunctions of conditions, and all disjunctive conditions.

**Definition 23 (legit conditions).** The class of *syntactically legit* (*legit*) conditions is defined inductively as follows.

(a) If $d$ is proper (generalized proper), then $d$ is syntactically legit (legit).

(b) If $ds = d_1, \dots, d_n$ is a sequence negative or positive conditions, then $\bigwedge_{i=1}^{n} d_i$ is syntactically legit. If $ds = d_1, \dots, d_n$ is a sequence of legit conditions and $ds$ is preserving, then $\bigwedge_{i=1}^{n} d_i$ is legit.

(c) If $d_i$ is syntactically legit (legit) for some $i$, then $\bigvee_{i=1}^{n} d_i$ is syntactically legit (legit).

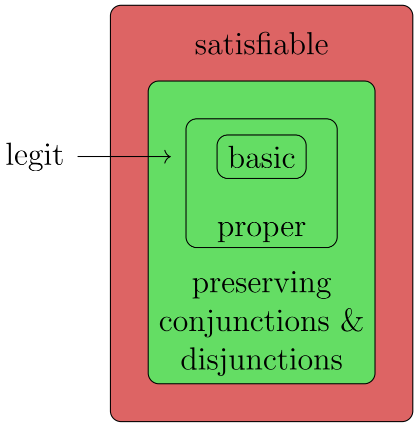


Figure 4.12: Hierarchy of classes of conditions

In Figure 4.12, all legit conditions (basic and proper conditions, preserving conjunctions and disjunctions) are marked in a green color. Satisfiable conditions that are not legit, are marked in red color.

**Remark.** The class of legit conditions properly includes the class of all syntactically legit conditions (see Example 43).

For legit conditions, repair programs can be constructed.

**Theorem 5.** For every legit condition, there is a solid (ad hoc) repair program.

**Example 43.** Consider the constraint $d = \forall(\boxed{\text{Pl}}, c_1 \wedge c_2)$ with the conditions $c_1 = \nexists(\boxed{\text{Pl}} \hookrightarrow \boxed{\text{Tk}} \xleftarrow{\text{tok}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}})$ and $c_2 = \exists(\boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}})$ meaning that for all Pl-nodes there do not exist two Tk-nodes with connecting containment tok-edges and there exists a Tk-node and a connecting containment tok-edge. The condition $c = c_1 \wedge c_2$ contains a conjunction. Consequently, it is not linear and by definition not proper. The condition $c$ is a generalized proper condition and there is a repair program for $c$, and thus, for the whole constraint $d$. The repair program $P_d$ for $d$ is constructed as follows: the constraint is of the form $\forall(a, c)$, with $a\colon \emptyset \hookrightarrow \boxed{\mathrm{Pl}}$, and by Theorem 2 $P_d = \langle \mathtt{Mark}(a, \neg c); P_c; \mathtt{Unmark}(a)\rangle\downarrow$. The condition $c$ is a generalized proper condition and there is a repair program $\langle P_{c_1}; P_{c_2}\rangle$ for $c$ with interface $\boxed{\mathrm{Pl}}$: by Lemma 17, there are repair programs $\mathcal{S}'\downarrow$ for $c_1$ and $\mathtt{try}\ \mathcal{R}$ for $c_2$, where $\mathcal{S} = \langle\, \boxed{\mathrm{Tk}} \xleftarrow{\mathrm{tok}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}} \Rightarrow \boxed{\mathrm{Tk}}\ \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}} \,\rangle$ and $\mathcal{R}$ is given in Example 32. Since the condition $c$ is conjunctive, it is a requirement that the program $P_{c_2}$ is $c_1$ preserving: first, all super numerous tok-edges are deleted at a marked Pl-node, afterwards, a tok-edge (together with a Tk-node if necessary) is added to the marked Pl-node. This is done for each Pl-node in a graph.

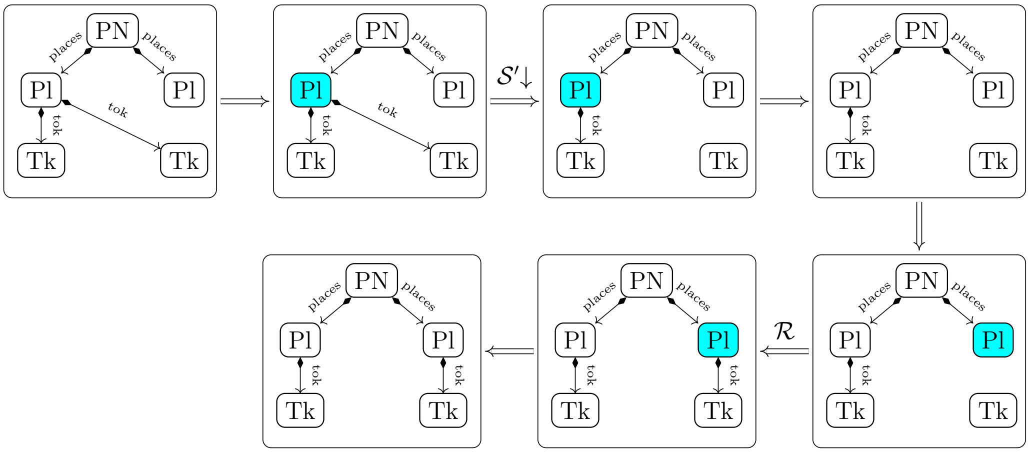


Figure 4.13: Transformation of the repair program for the legit condition

For the input graph in Figure 4.13, the program works as follows: the left Pl-node is marked because the Pl-node violates the condition $c_1$, i.e., the Pl-node has two tok-edges to the Tk-nodes. One of these two edges is deleted by $\mathcal{S}'\downarrow$. The Pl-node has exactly one tok-edge and satisfies the condition $c$, consequently, it is unmarked. The right Pl-node does not have a tok-edge, consequently, it is marked, the rule set $\mathcal{R}$ is applicable, and adds the missing tok-edge, finally it is unmarked. Afterwards, all Pl-nodes have exactly one tok-edge to a Tk-node, and the graph satisfies the constraint $d$.

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

(a) If $d$ is proper, then, by Theorem 2, there is a repair program for $d$. If $d$ is generalized proper, then $d$ is of the form $\mathrm{Q}(a, c)$ where $c$ is legit. By induction hypothesis, there is a repair program for $c$. By a generalized Theorem 2, there is a repair program for $d$. (b) Let $ds = d_1, \ldots, d_n$ be a sequence of legit conditions and $ds$ preserving. By induction hypothesis, there are repair programs for $d_1, \ldots, d_n$. By Theorem 3, there is a repair program for the conjunction $\bigwedge_{i=1}^{n} d_i$. (c) Let $d_i$ be legit. By induction hypothesis, there is a repair program for $d_i$. By Theorem 4, there is a repair program for $\bigvee_{i=1}^{n} d_i$. This completes the inductive proof. □

It turns out, that it is semi-decidable if a condition is generalized proper. For sequences of conditions, it is semi-decidable whether they are preserving, and it is semi-decidable whether a condition is legit.

**Lemma 22 (decidability/semi-decidability).**

(1) For conditions, it is decidable whether they are proper.
For conditions, is is semi-decidable whether they are generalized proper.

(2) Sequences of negative (positive) conditions are preserving.
For sequences of conditions it is semi-decidable whether they are preserving.

(3) For conditions, it is decidable whether they are syntactically legit.
For conditions, it is semi-decidable whether they are legit.

**Proof.** (1) Follows from the definition of proper conditions and the fact that it is semi-decidable whether a repair program can be constructed for an arbitrary satisfiable condition. (2) Follows from Fact 9 and the fact that for some sequences a preserving sequence can be found. (3) Follows from (1) and (2). □

## 4.6 Properties of the repair programs

We look for properties of the constructed repair programs. In more detail, we show that the solid repair programs for proper conditions are stable, maximally preserving, and terminating. Additionally, we show that the solid repair programs for legit conditions are stable and terminating. The proofs are based on induction over the structure of proper (legit) conditions.

### Stability

Given a graph satisfying the condition, the user may be interested that the repair program should not change the graph. A repair program that does not change the input graph, if the condition is already satisfied, is said to be stable.

**Definition 24.** A repair program $P$ for a condition $d$ is *stable* if, for all transformations $g \Rightarrow_P h$, $g \models d$ implies $g = h$.

The solid (ad hoc) repair programs are stable.

**Lemma 23 (stability).** Solid (ad hoc) repair programs for legit conditions are stable.

**Example 44.** Consider the constraint $\exists\, \boxed{\text{Pl}}$ and the repair programs $\texttt{try}\ \langle \emptyset \Rightarrow \boxed{\text{Pl}}\,, \nexists\, \boxed{\text{Pl}}\,\rangle$ or $\texttt{try}\ \langle \emptyset \Rightarrow \boxed{\text{Pl}}\quad \boxed{\text{Pl}}\,, \nexists\, \boxed{\text{Pl}}\,\rangle$. By the application condition $\nexists\, \boxed{\text{Pl}}$, both programs only change the graph, provided there does not exist a Pl-node.

**Proof.** $P_d$ is **stable**. By induction on the structure of legit conditions, we show that the repair programs are stable.

Let $d$ be a proper condition and $g \models d$.

(1) For $d = \texttt{true}$, $\texttt{Skip}$ is applicable with result $g$.
(2) For $d = \exists\, a$, by the application condition $\mathrm{ac} = \mathrm{Shift}(A \hookrightarrow B, \nexists\, a)$, $\mathcal{R}_a$ is not applicable and the application of $\texttt{try}\ \mathcal{R}_a$ yields $g$.
(3) For $d = \nexists\, a$, $\mathcal{S}'_a$ is applicable zero times, i.e., by the semantics of $\downarrow$ the result is $g$ (up to isomorphism).
(4) For $d = \exists\,(a, c)$, $P_d = \langle P_{\exists\, a}; \texttt{Mark}(a); P_c; \texttt{Unmark}(a)\rangle$. By induction hypothesis, $P_{\exists\, a}$ and $P_c$ are stable. Moreover, $\texttt{Mark}(a)$ and $\texttt{Unmark}(a)$ are stable. As a consequence, $P_d$ is stable.
(5) For $d = \forall(a, c)$, $P_d = \langle \texttt{Mark}(a, \neg c); P_c; \texttt{Unmark}(a)\rangle\downarrow$. By induction hypothesis, $P_c$ is stable. Moreover, $\texttt{Mark}(a, \neg c)$ and $\texttt{Unmark}(a)$ are stable. By the application condition $\neg c$ and the semantics of $\downarrow$, $P_d$ is stable.

If $d$ is generalized proper, then, by induction hypothesis, $P_d$ is stable.

For $d = \bigwedge_i^n d_i$, $P_d = \langle P'_1; \ldots; P'_n\rangle$ according to Theorem 3.3 where $P'_i = \langle P_i; P^{\mathrm{id}}_{\nexists a,i}\rangle$ and $P^{\mathrm{id}}_{\nexists a,i} = \langle \texttt{Mark}(a_i, \neg c_i); \mathcal{S}'^{\mathrm{id}}_{a,i}\rangle\downarrow$. By induction hypothesis, $P_i$ is stable, and, by Construction, $P^{\mathrm{id}}_{\nexists a,i}$ is only applicable, iff the condition is not satisfied. By the semantics of $\downarrow$, it is stable. For $d = \bigvee_i^n d_i$, then, by induction hypothesis, each $P_i$ is stable. Consequently, the programs $P_i$ and $\{P_1, \ldots, P_n\}$ are stable.

□

## Maximal Preservation

In the following, we look for a "maximally preserving" repair program, meaning, that the program preserves an input graph as long as there is no requirement for the non-existence of items. The resulting graph is a graph, that is “as close as possible” to the input graph. Whenever a condition requires the non-existence (existence) of a certain subgraph, there is no increasing (decreasing) repair program that repairs all graphs. Therefore, we look for maximally preserving repair programs. We start with maximally preserving repair programs for proper conditions and show that the solid repair programs for proper conditions are maximally preserving. Afterwards, we consider maximally preserving repair programs for conjunctive and disjunctive conditions. We give an approximation of the deleted number of graph elements for our repair programs.

For maximally preserving repair programs, we count the maximal number of elements that have to be deleted to satisfy a constraint for a given input graph. More general, we count the elements to be deleted for a morphism and a condition. A repair program is maximally preserving if for all transformation with the repair program, for which the number of preserved elements is greater than or equal to the number of size of the input morphism minus the maximal number of elements that have to be deleted. For that, we consider all transformations with the repair program and look for the preserved elements.

Formally, the definition of maximally preserving repair programs relies on the so-called track morphism. With every transformation $t\colon g \Rightarrow^* h$ a partial morphism can be associated that “follows” the items of $G$ through the transformation: this morphism is undefined for all items in $G$ that are removed by $t$, and maps all other items to the corresponding items in $H$.

**Definition 25 (track morphism).** The *track morphism* $\mathrm{tr}_{g \Rightarrow h}$ from $g\colon A \hookrightarrow G$ to $h\colon A \hookrightarrow H$ is the partial morphism defined by $\mathrm{tr}_{g\Rightarrow h}(x) = \mathrm{inc}_H(\mathrm{inc}_G^{-1}(x))$ if $x \in D$ and *undefined* otherwise, where $\mathrm{inc}_G = \mathrm{inc} \circ \mathrm{inc}_{G'}$ and $\mathrm{inc}_G^{-1}\colon \mathrm{inc}_G(D) \hookrightarrow D$ is the inverse of $\mathrm{inc}_G$. Given a transformation $g \Rightarrow^* h$, $\mathrm{tr}_{g\Rightarrow^* h}$ is defined by induction on the length of the transformation: $\mathrm{tr}_{g\Rightarrow^* h} = \mathrm{iso}$ for an isomorphism $\mathrm{iso}\colon G \to H$ from $G$ to $H$, and $\mathrm{tr}_{g\Rightarrow^* h} = \mathrm{tr}_{g'\Rightarrow h} \circ \mathrm{tr}_{g\Rightarrow^* g'}$ for $g \Rightarrow^* h = g \Rightarrow^* g' \Rightarrow h$.

**Definition 26 (maximally preserving repair for proper conditions).** A repair program $P_d$ for a proper condition $d$ over $A$ is *maximally preserving*, if, for all transformations $t\colon g \Rightarrow_{P_d} h$,

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

where, for a transformation $t$ via $P_d$, $\mathrm{pres}(P_d, t)$ denotes the number of preserved items by $t$, i.e., the items in the domain of the track morphism of $t$, and $\Delta(g, d)$ denotes the maximal number of necessary deletions.

Given an injective morphism $g\colon A \hookrightarrow G$ and a proper condition $d$ over $A$, $\Delta(g,d)$ is defined inductively as follows: $\Delta(g,\texttt{true}) = 0$, $\Delta(g, \exists\, a) = \ 0$,

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

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

**Informal Description.** Given a morphism $g\colon A \hookrightarrow G$ and a proper condition $d$ over $A$, we determine the maximal number of necessary deletions $\Delta(g,d)$. This is zero if the condition is `true` or of the form $\exists\, a$. For a condition of the form $\nexists\, a$, we consider all morphisms $g' \in \mathrm{Ext}(g)$, and sum up the number of deletions. For a condition of the form $\exists\, (a,c)$, we consider all morphisms $g' \in \mathrm{Ext}(g)$ and build the maximum of all $\Delta(g', c)$. For a condition of the form $\forall(a,c)$, we consider all morphisms $g' \in \mathrm{Ext}(g)$ and sum up the number of necessary deletions for the condition $c$ at that position, i.e., $\Delta(g', c)$.

**Example 45.** Consider the constraints $d = \exists\, (a, \nexists\, b)$ with $a\colon \emptyset \hookrightarrow \boxed{\text{Pl}}$ and $b\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}$, intuitively meaning there exists a Pl-node without a Tk-node and an edge between, and the input graph given in Figure 4.14. The condition $d$ is of the form $\exists\, (a,c)$, consequently, $\Delta(\emptyset \hookrightarrow^{g} G, d) = \min(g_i', \nexists\, b)$, where $g_i'$ with $i \in \{1,2\}$ are the morphisms $\boxed{\text{Pl}} \hookrightarrow G$. There are two places in $G$. Since $\nexists\, b$ is a negative condition, we sum up the elements, which have to be deleted, i.e., the tok-edge. For the left occurrence of the Pl-node, there is one tok-edge, i.e., $\mathrm{Ext}(g_1') = 1$, for the right one, there are two. Consequently, $\Delta(g,d) = \min(1,2) = 1$.

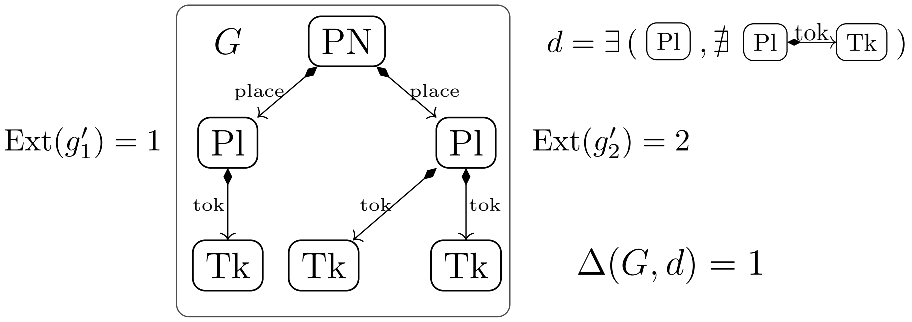
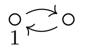
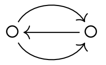
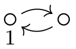
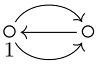
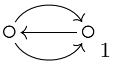

G
PN
place
place
Pl
Pl
tok
tok
tok
Tk
Tk
Tk
$d = \exists\, (\boxed{\text{Pl}}, \nexists\, \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}})$
$\mathrm{Ext}(g_1') = 1$
$\mathrm{Ext}(g_2') = 2$
$\Delta(G,d) = 1$

Figure 4.14: $\Delta(G,d)$

There are two transformations via the repair program $P_d$, the first one $t_1$ marks the left place, and deletes one tok-edge, i.e., $\text{pres}(P_d, t_1) = \text{size}(G){-}1 = \text{size}(G){-}\Delta(G, d)$, the other transformation $t_2$ marks the right place, and deletes two tok-edges, i.e., $\text{pres}(P_d, t_1) = \text{size}(G){-}2$. By Definition, the transformation $t_1$ is maximally preserving.

There are cases, where there exists a transformation with the repair program for proper conditions, which preserves more elements, than the counted least number of elements, which have to be deleted. This may be seen in the following example.

**Example 46** **($\text{pres}(P_d, t) > \Delta(G, d)$).** Let $d = \exists(\ \underset{1}{\circ}\ , \nexists$  ). Consider the input graph $G =$  , more formally, the input morphism $g\colon \emptyset \hookrightarrow G$. Since $d$ is of the form $\exists\,(a, c)$, with $a\colon \emptyset \hookrightarrow \underset{1}{\circ}$ , $c = \nexists\, b$, and $b\colon \underset{1}{\circ} \hookrightarrow$  , $\Delta(g, d) = \min_{g' \in \text{Ext}(g)}$. For both morphisms $g'_i$ from $\underset{1}{\circ}$ to one of the nodes in $G$, i.e., $g'_1\colon \underset{1}{\circ} \hookrightarrow$  and $g'_2\colon \underset{1}{\circ} \hookrightarrow$  there are two extensions $g''$ from $b$ to $g'$, such that $g'' \circ b = g'$, thus $\Delta(g'_i, \nexists\, b) = 2$ and $\Delta(g, d) = \min(2, 2) = 2$. There are transformations $t : g \Rightarrow_{P_d} h$ to a morphism $h$ satisfying $d$ with $\text{pres}(P_d, t) = \text{size}(g){-}2$ as well as one with $\text{pres}(P_d, t') = \text{size}(g){-}1$, i.e., $\text{pres}(P_d, t) = \text{size}(g){-}1 > \text{size}(g){-}\Delta(g, d)$.

**Lemma 24 (maximal preservation).** The solid repair programs for proper conditions are maximally preserving.

**Informal Description.** Intuitively, the solid repair programs for proper conditions are maximally preserving because of the following.

(1) For the condition `true`, nothing is to do.

(2) For positive conditions $\exists\, a$, we only add nodes and edges.

(3) For negative conditions $\nexists\, a$, we count the number of elements that have to be deleted. In more detail, for an occurrence of the condition, we count the number of violations at that position. For that, we count all edges which have to be deleted and, if the morphism $a$ is not edge-increasing, we count the number of nodes, which have to be deleted and the dangling edges for each of the nodes. We choose the maximal number of dangling edges, which have to be deleted. The negative condition cannot be satisfied otherwise.

(4) For existential conditions $\exists\,(a, c)$, we select the occurrence, where we have to delete the maximal number of violations of the subcondition $c$.

(5) For universal condition $\forall(a, c)$, we consider each violation of the subcondition $c$ and build the sum of all of them. If the proper condition ends with `true`, this sums up to 0.

**Proof (of Lemma 24).** The maximal preservation of the repair program $P_d$ is shown by induction of the length of transformations: We show that for all transformations $t\colon g \Rightarrow_{P_d} h$,

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

using the fact that, if $P$ is maximally preserving, then $P' = \langle \texttt{Mark}(a); P; \texttt{Unmark}(a) \rangle$ is maximally preserving.

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

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

(2) For $d = \exists\, a$, $P_d = \texttt{try}\ \mathcal{R}_a$, and $\text{pres}(\texttt{try}\ \mathcal{R}_a, t) = \text{size}(G)$.

(3) For $d = \nexists\, a$, $P_d = \mathcal{S}'_a\!\downarrow$. (a) If $g \models d$, then $\text{pres}(\mathcal{S}'_a\!\downarrow, t) = \text{size}(G)$. (b) If $g \not\models d$, then the transformation $g \Rightarrow^{+}_{\mathcal{S}'_a\downarrow} h$ is of the form $g \Rightarrow_{\mathcal{S}'_a} g_1 \Rightarrow_{\mathcal{S}'_a\downarrow} h$ where $t_1$ denotes the transformation starting with $g_1$. For a one-step transformation, the number of repairs is bounded by the number of extensions $g'$ of $g$, added with the number of dangling edges at that position, i.e., (*) $\text{size}(G_1) = \text{size}(G) - \Delta(g', d)$ with $\Delta(g', d) = 1 + \mathit{dang}(g')$. By definition of $\Delta$, (**) $\Delta(g, d) = \Delta(g', d) + \Delta(g_1, d)$. Applying this for arbitrary finite transformations, the deletions are bounded by the number $g'$ of extensions of $g$, i.e., (**) $\sum_{g' \in \text{Ext}(g)} (1 + \mathit{dang}(g'))$. The solid repair program deletes for each extension $g' \in \text{Ext}(g)$ exactly one edge, if the morphism is edge-increasing, and one node plus the number of dangling edges, i.e., (***) $\sum_{g' \in \text{Ext}(g)} (1 + \mathit{dang}(g'))$.

$$\begin{array}{rcll} \text{pres}(P_d, t) & \geq & \text{size}(G) - \sum_{g' \in \text{Ext}(g)} (1 + \mathit{dang}(g')) & (\text{***}) \\ & \geq & \text{size}(G) - \sum_{g' \in \text{Ext}(g)} (1 + \mathit{dang}(g')) & (\text{**}) \\ & = & \text{size}(G) - \Delta(g, d) & (\text{Def. of } \Delta) \end{array}$$

(4) For $d = \exists\,(a, c)$, $P_d = \langle P_{\exists\, a}; \langle \texttt{Mark}(a); P_c; \texttt{Unmark}(a) \rangle\rangle$. (a) If $g \models d$, then $\text{pres}(P_d, t) = \text{size}(G)$. (b) If $g \not\models d$, then $g \Rightarrow_{P_d} h$ is of the form $g \Rightarrow_{P_{\exists\, a}} g_1 \Rightarrow_{P'_c} h$. Then, since $P_{\exists\, a}$ is increasing (*) $\text{size}(G_1) \geq \text{size}(G)$. For all $g' \in \text{Ext}(g)$, the number of deletions is bounded by the maximal number of deletions of the program $P_c$, i.e.,
(**) $\Delta(g, d) = \max_{g' \in \text{Ext}(g)} (\Delta(g', c))$. By induction hypothesis, the program $P_c$ is maximally preserving, i.e, $\Delta(g', c)$ is maximal.

$$\begin{array}{rcll} \text{pres}(P_d, t) & = & \text{pres}(P'_c, t_1) & (\text{*}) \\ & \geq & \text{size}(G_1) - \max_{g' \in \text{Ext}(g)} (\Delta(g', c)) & (\text{ind hyp}) \\ & = & \text{size}(G) - \Delta(g, d) & (\text{**}) \end{array}$$

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

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

This completes the inductive proof. □

It remains the question, how maximal preservation for conjunctive and disjunctive conditions can be defined. One may assume, that for a conjunctive condition, the number of deleted elements can be added, i.e., for $d = d_1 \wedge d_2$ define $\Delta$ as $\Delta(g,d_1) + \Delta(g,d_2)$. Unfortunately, this is not so easy. There are cases, where the condition cannot be satisfied without deleting more elements. There are conditions, where we have to delete the whole input graph in the worst case. This may be seen in the following Example 47.

**Example 47 (deleting repair program).** Consider the conjunction $d = d_1 \wedge\ d_2 = \nexists\ \circ\!\!\curvearrowright\ \wedge \forall(\ \circ\ , \exists\ \circ\!\!\curvearrowright\ )$, intuitively meaning, there does not exist a node with a loop, and for all nodes, there exists a loop. The conjunction $d$ is equivalent to $\nexists\ \circ$ .

In the following, we consider the conjunction $d$ and not the equivalence. By the Repair Theorem II, there are repair programs $P_1 = \langle\ \circ\!\!\curvearrowright\ \Rightarrow\ \circ\ \rangle\downarrow$ for $d_1$ and $P_2 = \langle\texttt{Mark}(a,\neg c); \texttt{try}\ \mathcal{R}^{d_1}; \texttt{Unmark}(a)\rangle\downarrow$ for $d_2$, where $a\colon \emptyset \hookrightarrow\ \circ$ , $c = \exists(\ \underset{1}{\circ}\ \hookrightarrow\ \underset{1}{\circ}\!\!\curvearrowright\ )$, and $\mathcal{R} = \langle\ \underset{1}{\circ}\ \Rightarrow\ \underset{1}{\circ}\!\!\curvearrowright\ , \nexists\ \underset{1}{\circ}\!\!\curvearrowright\ \rangle$ (note that, for simplicity, the interfaces of the rules are omitted). The program $P_2$ is not $d_1$-preserving. Consequently we construct the $d_1$-preserving version of $P_2$, where the rule in $\mathcal{R}$ is equipped by the application condition $\nexists\ \underset{1}{\circ}$ . In general, the $d_1$-preserving version of $P_2$ is not a repair program for $d_2$, consequently we consider the $d_1$-preserving repair program $P'^{d_1}_2 = \langle P^{d_1}_2; P^{\mathrm{id}}_{\nexists a}\rangle$, followed by the decreasing repair program $P^{\mathrm{id}}_{\nexists a} = \langle\texttt{Mark}(a,\neg c); \langle\ \underset{1}{\circ}\ \Rightarrow \emptyset\rangle'\rangle\downarrow$.

The program intuitively works as follows: By $P_1$, all loops in a graph are deleted. Then, $P^{d_1}_2$ selects a node without a loop. By the $d_1$-preserving application condition $\nexists\ \underset{1}{\circ}$ , the rule (set) $\mathcal{R}$ is not applicable. Consequently, the program $P^{\mathrm{id}}_{\nexists a}$ deletes all nodes, which do not have a loop, i.e., all nodes.

For an input graph $G$, or more formally the input morphism $g\colon \emptyset \hookrightarrow G$, $\Delta(g, d_1)$ depends on the number of occurrences of nodes with a loop in $G$. For each input graph $g$, $\Delta(g, d_2) = 0$ because we do not have to delete any loops to satisfy $d_2$. However, the repair program cannot repair the whole conjunction without deleting the whole input graph, i.e., $\Delta(g, d) = \text{size}(g)$.

For a disjunctive condition, the number of deleted elements depends on the input graph and the chosen program. This may be seen in the following Example 48.

**Example 48 (repair for disjunctive conditions).** Consider the disjunction $d = d_1 \vee\ d_2 = \nexists\ \circ\!\circlearrowright\ \vee \nexists\ \circ\!\rightarrow\!\circ$ , intuitively meaning, there does not exist a node with a loop or there does not exist two nodes and a real edge, i.e., not a loop. in between. By the Repair Theorem I, there are repair programs $P_1 = \langle\ \circ\!\circlearrowright\ \Rightarrow\ \circ\ \rangle\!\downarrow$ and $P_2 = \langle\ \circ\!\rightarrow\!\circ\ \Rightarrow\ \circ\quad\circ\ \rangle\!\downarrow$ for $d_1$ and $d_2$, respectively. By the Repair Theorem III, the programs $P_1$, $P_2$ and $P_d = \{P_1; P_2\}$ are repair programs for $d$. The number of deleted elements depends on the input graph and the chosen repair program: if the input graph contains more loops than edges, then the program $P_2$ can be applied to delete the edges instead of loops. Vice versa, if the graph contains more edges than loops, the program $P_1$ can be applied to delete the loops instead of the edges.

However, we can give a worst-case approximation of the number of graph elements, which are deleted by the solid repair programs for conjunctive and disjunctive conditions. For conjunctions of negative (or universal and preserving) conditions, we can simply add the number of deleted elements for each of the subconditions. For conjunctive conditions, where the first condition is negative, and the second condition is universal, i.e., $d_2 = \forall(a, c)$, in general, there are cases where we have to delete occurrences of the morphism $a$ violating the condition $c$. Consequently, for the program $P^{\text{id}}_{\nexists a}$ we approximate the number of deletions for the worst case, where the minimal number of all occurrences of the morphism $a$ are deleted, i.e., $\Delta(g, \nexists\, a)$. For disjunctive conditions, we choose the maximal number of elements for the subprograms.

**Lemma 25.** The maximal number of deleted elements of the solid repair programs for conjunctive conditions is

$$\begin{aligned}
\Delta(g, d_1 \wedge d_2) &= \Delta(g, d_1) + \begin{cases} \Delta(g, d_2) & \text{if } d_1, d_2 \text{ negative (universal \& preserving)} \\ \Delta(g, \nexists\, a) & \text{if } d_1 \text{ negative, } d_2 = \forall(a, c) \end{cases} \\
\Delta(g, d_1 \vee d_2) &= \max\{\Delta(g, d_1), \Delta(g, d_2)\}
\end{aligned}$$

**Proof.** For simplicity, we proof the statement for $n = 2$. For $n := n+1$ the proof is similar. 1. Let $d = d_1 \wedge d_2$ be a conjunctive condition, $P_d = \langle P_1; P_2^{d_1}; P_{\nexists a}^{\mathrm{id}} \rangle$ the repair program according to Theorem 3, and $g \Rightarrow_{P_d} h$ be a transformation.

(a) If $g \models d$, then $\mathrm{pres}(P_d, t) = \mathrm{size}(g)$. (b) If $g \not\models d$, then $g \Rightarrow_{P_d} h$ is of the form $g \Rightarrow_{P_1} g_1 \Rightarrow_{P_2^{d_1}} g_2 \Rightarrow_{P_{\nexists a}^{\mathrm{id}}} h$, where $t_i$ is the transformation starting at $g_i$ ($i = 1, 2$). By induction hypothesis, $P_1$ is maximally preserving, i.e., $\mathrm{pres}(P_1, t) \geq \mathrm{size}(g) - \Delta(g, d_1)$. If $g_1 \not\models d_2$, there are three cases:

If $d_2$ is negative, then, by induction hypothesis, $P_2$ is maximally preserving, and $\mathrm{pres}(P_2, t_1) \geq \mathrm{size}(g_1) - \Delta(g_1, d_2)$. (Since $d_1, d_2$ are negative, $P_{\nexists a}^{\mathrm{id}}$ is empty.) If $d_1, d_2$ are universal, then, by induction hypothesis, $P_2$ is maximally preserving, i.e., $\Delta(g_1, d_2) = 0$, and $\mathrm{pres}(P_2, t_1) \geq \mathrm{size}(g_1) - \Delta(g_1, d_2)$. By assumption, $P_2$ is $d_1$-preserving. Consequently, $P_{\nexists a}^{\mathrm{id}}$ is not applicable, $g_2 \cong h$, $\mathrm{size}(g_2) = \mathrm{size}(h)$, and $\mathrm{pres}(P_{\nexists a}^{\mathrm{id}}, t_2) \geq \mathrm{size}(g_2) - \Delta(g_2, d_2)$. If $d_2$ is universal, $P_2^{d_1}$ is increasing, and by induction hypothesis, $\Delta(g_1, d_2) = 0$, $\mathrm{pres}(P_2^{d_1}, t_1) \geq \mathrm{size}(g_1) - \Delta(g_1, d_2)$. By induction hypothesis, $P_{\nexists a}^{\mathrm{id}}$ is maximally preserving, i.e., $\Delta(g_2, \nexists a) = \sum_{g_2' \in \mathrm{Ext}(g_2)}$ $(dang(g_2') + 1) \mid g_2' \not\models c$, and $\mathrm{pres}(P_{\nexists a}^{\mathrm{id}}, t_2) \geq \mathrm{size}(g_2) - \Delta(g_2, \nexists a)$.

2. Let $d = d_1 \vee d_2$ be a disjunctive condition, the programs $P_1$, $P_2$, or $P_d = \{P_1, P_2\}$ be the repair programs according to Theorem 4, and $g \Rightarrow_{P_d} h$ be a transformation. (a) If $g \models d$, then $\mathrm{pres}(P_d, t) = \mathrm{size}(g)$. (b) If $g \not\models d$, then $g \Rightarrow_{P_d} h$ is of the form $h_1 \Leftarrow_{P_1} g \Rightarrow_{P_2} h_2$. By induction hypothesis, $P_1$ and $P_2$ are maximally preserving, i.e., $\mathrm{pres}(P_d, t) \geq \mathrm{size}(g) - \max\{\Delta(g, d_1), \Delta(g, d_2)\}$. □

To get maximally preserving repair programs, we have to make sure that the underlying repairing sets are maximally preserving. If a rule set is maximally preserving, then the number of deleted items for negative conditions $\nexists(\, A \hookrightarrow C\,)$ is minimal. If a rule set is decreasing, edges are deleted instead of nodes, whenever possible, since it is more costly to delete nodes than edges. For the ad hoc repair programs this does not hold: A rule $C \Rightarrow A$ deletes $[\mathrm{size}(C) - \mathrm{size}(A)]$ items, although only one item has to be deleted, i.e., in general, it is not maximally preserving.

**Fact 14.** In general, the ad hoc repair programs are not maximally preserving.

**Remark.** Often, deletions can be avoided when edges are redirected to some other node (see, e.g. Nassar [Nas20]). This is a further topic of investigation.

## Termination

The repair program is *terminating*, if the relation $\Rightarrow$ (see Definition 8) is terminating, i.e., there is no infinite transformation with the program. The solid repair programs for legit conditions are terminating.

**Lemma 26 (termination).** The solid repair programs for legit conditions are terminating.

**Informal Description.** The solid repair programs for legit conditions are terminating because of the following.

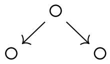
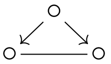
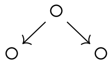
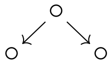
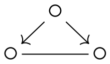
For positive conditions $\exists\, a$, we only add nodes and edges, provided that they do not exist. Consequently, the number of added nodes and edges is bounded by the size of the input or the condition.

For universal conditions $\forall(a, c)$, the number of nodes and edges is bounded, and by the $\texttt{Mark}(a, \neg c)$ statement, each non-repaired occurrence of $a$ can only be marked at most once. Contradiction to the assumption of an infinite transformation.

The solid repair programs for conjunctive conditions are terminating because, by induction hypothesis, each subprogram is terminating and the preserving repair program is decreasing.

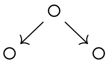
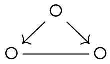
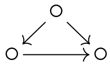
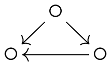
The solid repair programs for disjunctive conditions are terminating, provided that the repair program for the subconditions is terminating.

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

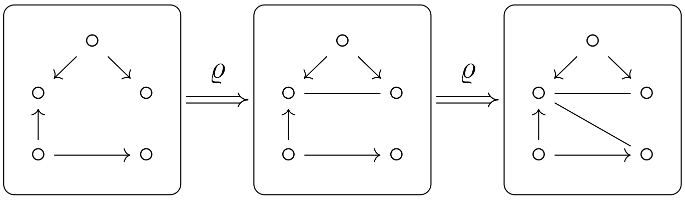
**Example 49.** Let $d = \forall($  $, \exists$  $)$[7], with $a\colon \emptyset \hookrightarrow$  and $c = \exists\,($  $\hookrightarrow$  $)$. Then, the solid repair program

$$P_d = \langle \texttt{Mark}(a, \neg c); \texttt{try } \mathcal{R}; \texttt{Unmark}(a)\rangle\downarrow$$

contains the rule

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

Applying the program to the graph below, we obtain the following transformation:[8]

$G =$ 

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

[8] For simplicity, we omit the marking and unmarking of elements, in the transformation.

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

**Proof.** $P_d$ is **terminating**. By induction on the structure of $d$ using some elementary rules for concluding termination.

1. Finite rule sets are terminating.
2. If $P$ is terminating, then $P' = \langle \texttt{Mark}(a); P; \texttt{Unmark}(a)\rangle$ is terminating.
3. If programs $P, Q$ are terminating, then $\langle P; Q\rangle$ is terminating.
4. If a rule set $\mathcal{S}$ is decreasing, then $\mathcal{S}\downarrow$ is terminating.

Let $d$ be a proper condition, $P_d$ be the corresponding program, $g \Rightarrow_{P_d} h$, $g\colon A \hookrightarrow G$, $h\colon A \hookrightarrow H$. Let $m$ denote the number of nodes of the largest graph of $d$, $n = \max(m, |\mathrm{V}_G|)$ the maximal number of nodes in $d$ or $G$, and $k$ be the maximal number of parallel edges in $d$. Let $K_n^k$ denote the complete graph over $A$ with $n$ nodes and, for each pair of nodes $\langle v_1, v_2\rangle$, there are $k$ parallel edges from $v_1$ to $v_2$. We show: $H \sqsubseteq K_n^k$ (i.e., there is an injective morphism from $H$ into $K_n^k$.) From this it follows that $P_d$ is terminating because the existence of a finite graph contradicts the assumption of the existence of an infinite transformation.

(1) For $d = \texttt{true}$, $P_d = \texttt{Skip}$ is terminating, and $H \cong G \sqsubseteq K_n^k$.

(2) For $d = \exists\, a$, $P_d = \texttt{try}\ \mathcal{R}_a$ is terminating, and $H \sqsubseteq K_n^k$. This may be seen as follows: Let $g \Rightarrow_\varrho h$, be a direct transformation through the rule $\varrho = \langle b, B \Rightarrow C, \mathrm{ac}_B \wedge \mathrm{ac}, a\rangle$ in $\mathcal{R}_a$. If $|\mathrm{V}_G| < |\mathrm{V}_C|$, then $|\mathrm{V}_B| = |\mathrm{V}_G|$ and, by the application condition $\mathrm{ac}_B$ (there is no larger $B'$ with $B \subset B' \sqsubseteq C$), $|\mathrm{V}_H| = |\mathrm{V}_C|$. If $|\mathrm{V}_G| \geq |\mathrm{V}_C|$, then $|\mathrm{V}_B| \leq |\mathrm{V}_C|$ and $|\mathrm{V}_H| = |\mathrm{V}_G|$. Thus $|\mathrm{V}_H| = \max(m, |\mathrm{V}_G|) = n$. Moreover, by the application condition $\mathrm{ac}_B$, $|\mathrm{E}_H| \leq k \cdot |\mathrm{V}_H| \times |\mathrm{V}_H|$ where $k$ is the maximal $k$ parallel edges in $\mathcal{R}_a$.

(3) For $d = \nexists\, a$, $\mathcal{S}_a$ is decreasing, $P_d = \mathcal{S}'_a\downarrow$ is terminating, and $H \sqsubseteq K_n^k$.

(4) For $d = \exists\,(a, c)$, $P_d = P_{\exists\, a}; \langle \texttt{Mark}(a); P_c; \texttt{Unmark}(a)\rangle$ is terminating, and $H \sqsubseteq K_n^k$. By induction hypothesis, $P_{\exists\, a}$, $P_c$, and, thus, $P_d$ is terminating. (The application of $\texttt{Mark}(a)$ and $\texttt{Unmark}(a)$ do not change the size.) Then, for $g \Rightarrow_{P_{\exists\, a}} g'$, $G' \sqsubseteq K_n^k$, and for $g' \Rightarrow_{P_c} h$, $H \sqsubseteq K_n^k$. Consequently, for $g \Rightarrow_{P_d} h$, $H \sqsubseteq K_n^k$.

(5) For $d = \forall(a, c)$, $P_d = \langle \texttt{Mark}(a, \neg c); P_c; \texttt{Unmark}(a)\rangle$ is terminating, and $H \sqsubseteq K_n^k$. By induction hypothesis, $P_c$ is terminating and, since the application of $\texttt{Mark}(a, \neg c)$ and $\texttt{Unmark}(a)$ do not change the size, the program
$P'_c = \langle \texttt{Mark}(a, \neg c); P_c; \texttt{Unmark}(a)\rangle$ is terminating. Moreover, for all transformations $g_i \Rightarrow_{P'_c} g_{i+1} \Rightarrow_{P'_c} \ldots \Rightarrow_{P'_c} h$, $H \sqsubseteq K_n^k$. Since the condition $d$ is universal, $d$ ends with $\texttt{true}$ and the program $P_d$ is increasing. Suppose $\Rightarrow$ is not terminating. Then,

there must be a graph with infinite number of nodes or edges. Contradiction to $H \sqsubseteq K^k_n$. By `Mark`$(a, \neg c)$, it is not possible to apply $P_c$ at a repaired position. Consequently, for proper conditions, the solid repair program $P_d$ is terminating.

For generalized proper conditions, the proof is similar: If $d$ is generalized proper, then, by induction hypothesis, $P_d$ is terminating.

For $d = \bigwedge^n_{i=1} d_i$, $P_d = \langle Q_1; Q'^{e_1}_2 \rangle$, with $Q_1 = \langle P_1; \ldots; P_k \rangle$ and $Q'^{e_1}_2 = \langle P'^{e_1}_{k+1}; \ldots; P'^{e_1}_n \rangle$, where $P'^{e_1}_j = \langle P^{e_1}_j; P^{\mathrm{id}}_{\nexists a,j} \rangle$. By induction hypothesis, $P_i$ is terminating, consequently, $Q_1 = \langle P_1; \ldots; P_k \rangle$ is terminating. Furthermore, $P'^{e_1}_j$ is terminating: By induction hypothesis, $P^{e_1}_j$ is terminating. By Construction, $P^{\mathrm{id}}_{\nexists a,j}$ is decreasing, consequently it is terminating. By the termination of the programs, the sequential composition is terminating, consequently, $P'^{e_1}_j$ is terminating. Finally $Q'^{e_1}_2 = \langle P'^{e_1}_{k+1}; \ldots; P'^{e_1}_n \rangle$ is terminating: by the termination of each $P'^{e_1}_j$, the sequential composition is terminating. Consequently, $\langle Q_1; Q'^{e_1}_2 \rangle$ is terminating.

For $d = \bigvee^n_{i=1} d_i$, by induction hypothesis, some $P_i$ is terminating. Consequently, $\{P_1, \ldots, P_n\}$ is terminating.

This completes the inductive proof. □

To get terminating repair programs, we have to make sure that the underlying repairing sets do not introduce new occurrences. If a program is terminating, then the number of added items for universal conditions is minimal. If a rule set is increasing, new occurrences of violation may be introduced. For the ad hoc repair programs this does not hold: A rule $A \Rightarrow C$ adds [size($C$)−size($A$)] items. This rule, applied as long as possible, may yield to non-terminating repair programs.

**Fact 15.** In general, the ad hoc repair programs are not terminating.

**Example 50.** For the constraint $d = \forall(\ \underset{1}{\circ}\ , \exists\ \underset{1}{\circ}\!\longrightarrow\!\circ\ )$, the program $P_d = \langle\ \underset{1}{\circ} \Rightarrow \underset{1}{\circ}\!\longrightarrow\!\circ\ , \nexists\ \underset{1}{\circ}\!\longrightarrow\!\circ\ \rangle\!\downarrow$ is an ad hoc repair program which does not create cycles. The program is not terminating: For each node without an edge it adds a node together with an edge. The newly created node does not possess an edge and the program is applicable at that node again. Consequently, it is not terminating.

## 4.7 Grammar-based repair

In this section, we consider the problem of *grammar-based graph repair*[9], i.e., programs which are based on a given set of rules. Consider the main idea of our

[9] In [SH19], this is called rule-based repair because the rule set $\mathcal{R}$ is given. On the other hand, every repair program may be seen as rule-based because they are composed by rules. Thus, in the following, we call the repair as grammar-based.

approach again, shown in Figure 4.1. The first step is to translate the structure of the meta-model and its typing into a (typed) graph grammar. This has the advantage that the grammar generates exactly those (typed) graphs representing the structure of the models. Up to now, there was a problem: in general, the resulting (typed) graphs might not be in the language of the grammar, and consequently, do not correspond to the structure of the meta-model. This is the motivation for the next section, in which we consider repair programs that are based on a set of rules. This has the advantage that for each transformation via a grammar-based program, there is a transformation via the rules of the grammar. If a graph is generated by a grammar with rule set $\mathcal{R}$, then, after the application of an $GG$-based program, the result can be generated by the grammar, too. Each transformation with the program corresponds to the structure of the meta-model. This is interesting in contexts where the language is defined by a grammar, like triple graph grammars in the sense of Schürr [Sch94].

The problem of *grammar-based graph repair* is as follows: Given a set of rules $\mathcal{R}$ and a constraint $d$, try to construct a repair program $P$ based on the rule set $\mathcal{R}$, i.e., we allow to equip the rules of $\mathcal{R}$ with the dangling-edges operator (see Section 2.3), context (see Definition 28), application conditions [HP09], and interfaces [Pen09].

**Grammar-based repair problem**

**Given:** A grammar $GG = (\mathcal{R}, S)$ and a graph constraint $d$.
**Task:** Try to find an "$GG$-based" repair program $P$ for $d$.

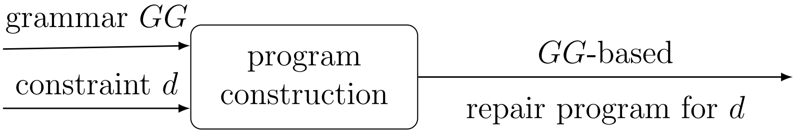


Figure 4.15: Idea of a grammar-based repair program

A grammar-based program is a program based on a set of rules equipped with the dangling-edges operator, context (see Definition 28), application condition, and interface.

**Definition 27 (grammar-based programs).** Given a graph grammar with rule set $\mathcal{R}$, a program is *GG-based*, if all rules in the program are rules in $\mathcal{R}$ equipped with dangling-edges operator, context (see Definition 28), application condition, and interface. Additionally, the program `Skip` is $GG$-based. In more detail, a program is *GG-based*, if all rules in the program belong to the set $\text{ggb}(\mathcal{R})$, inductively defined as follows.

(1) All rules in $\mathcal{R}$ are in ggb($\mathcal{R}$).

(2) If $\varrho \in \text{ggb}(\mathcal{R})$ with application condition ac, then $\langle \varrho, \text{ac}\rangle$ is in ggb($\mathcal{R}$).

(3) If $\varrho \in \text{ggb}(\mathcal{R})$ with kontext morphism $k\colon K \hookrightarrow K'$, then $\varrho^k$ is in ggb($\mathcal{R}$).

(4) If $\varrho \in \text{ggb}(\mathcal{R})$ with interfaces $x$ and $y$, then $\langle x, \varrho, y\rangle$ is in ggb($\mathcal{R}$).

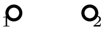
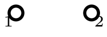
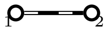
(5) If $\varrho \in \text{ggb}(\mathcal{R})$, then $\varrho'$ is in ggb($\mathcal{R}$).

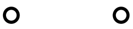
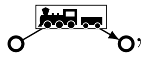
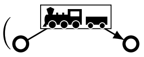
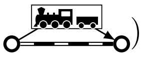
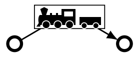
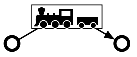
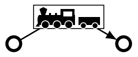
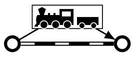
**Example 51.** The rule $\texttt{AddTok} = \langle\ \boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}\ \rangle$ equipped with interface morphisms $x_1\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}$ and $y_1\colon \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}$, and equipped with the application condition $\text{ac} = \nexists(\ \boxed{\text{Pl}} \hookrightarrow \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}\ )$ yields the $\{\texttt{AddTok}\}$-based program $\texttt{try AddTok}'$, where $\texttt{AddTok}' = \langle x_1, \texttt{AddTok}, \text{ac}, y_1\rangle$. At a marked Pl-node, the program tries to add a Tk-node together with a connecting containment tok-edge, provided it does not exist one. The rule $\texttt{AddTok}'$ can be used as part of the grammar-based repair program for the constraint $\forall(\ \boxed{\text{Pl}}\ , \exists\ \boxed{\text{Pl}}\xrightarrow{\text{tok}}\boxed{\text{Tk}}\ )$.

To get a grammar-based repair program for a condition (see Definition 27), we sometimes have to equip the rules with a morphism, called context. The reason for this are the interfaces of the rules: by construction, each rule has interfaces $A$. In general, there are rules, which might be used to repair a condition but does not have an occurrence of the graph $A$ in the left-hand side. In that case, we equip the rule with a context morphism. A rule equipped with context morphism describes how a smaller rule can be extended to a larger rule, the rule with context.

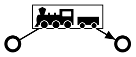
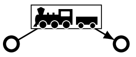
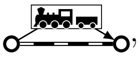
**Definition 28 (rules with context).** For a rule $\varrho = \langle x, p, \text{ac}, y\rangle$ and a morphism $k\colon K \hookrightarrow K'$, $\varrho^k = \langle x', p', \text{ac}', y'\rangle$ denotes the rule *equipped with context* where $p' = \langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ is the derived rule, $L'$ and $R'$ are the pushout objects of $L \hookleftarrow K \hookrightarrow K'$ and $R \hookleftarrow K \hookrightarrow K'$ (see (1) and (2) in the diagrams below), respectively, $l\colon L \hookrightarrow L'$ and $r\colon R \hookrightarrow R'$ are the corresponding morphisms, $\text{ac}' = \text{Shift}(l, \text{ac})$, $x' = l \circ x$, and $y' = r \circ y$.

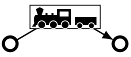
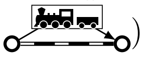
$$\begin{tikzcd}
\text{ac} \rhd L \arrow[d, hook, dashed, "l"'] \arrow[dr, phantom, "(1)"] & & K \arrow[ll, hook'] \arrow[rr, hook] \arrow[d, hook, "k"'] \arrow[dr, phantom, "(2)"] & & R \arrow[d, hook, dashed, "r"] \\
\text{ac}' \rhd L' & & K' \arrow[ll, hook', dashed] \arrow[rr, hook, dashed] & & R'
\end{tikzcd}$$

**Example 52.** The rule `Build` $= \langle$  $\hookleftarrow$  $\hookrightarrow$  $\rangle$ equipped with

- context $k'\colon$  $\hookrightarrow$ ,
- application condition ac $= \nexists\,($ $\hookrightarrow$ ), and
- interface morphisms $x\colon$  $\hookrightarrow$  and $y\colon$  $\hookrightarrow$ 

yields the rule

`Build2` $= \langle x,$  $\hookleftarrow$  $\hookrightarrow$ , ac, $y\rangle$.

At a marked train, the rule adds a piece of track, provided there does not exist one. The rule `Build2` can be used as part of the grammar-based repair program for the constraint $\forall($, $\exists$ ).

**Remark.** To restrict the applicability of the grammar-based program to previously marked elements, the rules are equipped with interface. For this, we sometimes have to equip the rules with a morphism, called context. The reason for this are the interfaces of the rules: by construction, each rule has interfaces $A$. In general, there are rules, which might be used to repair a condition, but does not have an occurrence of the graph $A$ in the left-hand side. In that case, we equip the rule by a context morphism.

**Fact 16.** The equipment of the rules with a context or an application condition does not influence the result: Whenever a rule with context or application condition is applied, then the corresponding underlying rule in $\mathcal{R}$ can be applied.

1. For all rules $\varrho^k$, $G \Rightarrow_{\varrho^k} H$ implies $G \Rightarrow_{\varrho} H$.
2. For all rules $\langle\varrho, \mathrm{ac}\rangle$, $G \Rightarrow_{\langle\varrho,\mathrm{ac}\rangle} H$ implies $G \Rightarrow_{\varrho} H$.

where $\varrho^k$ denotes the rule $\varrho$ equipped with context $k\colon K \hookrightarrow K'$, and $\langle\varrho, \mathrm{ac}\rangle$ the rule $\varrho$ equipped with application condition ac.

**Idea.** Given a graph grammar $GG = (\mathcal{R}, S)$, the construction of an $GG$-based repair program for a condition $d$ is based on the following idea.

(1) Try to construct a repair program for the condition $d$.

(2) Try to refine the rules of the repair program (for $d$) by equivalent transformations via the given rule set of $GG$

(3) Transform the transformations into equivalent $GG$-based programs.

(4) Replace each repairing set in $P_d$ by an equivalent $GG$-based program.

The proceeding is illustrated in Figure 4.16. The first step (1), is to construct a repair program for the condition. For illustration, the condition in the figure is a proper condition. This yields a repair program, which contains for each existential subcondition the corresponding rule sets ($\mathcal{R}_{a_1}$ and $\mathcal{R}_{a_3}$). The second step (2), is the refinement of these rule sets, by equivalent transformations via the rules of the input graph grammar. In the third step (3), we transform these transformations into equivalent programs ($P_{a_1}$ and $P_{a_3}$), which are based on the graph grammar. In the fourth step (4), we replace each of the repairing sets of the repair program by the equivalent graph grammar based programs, i.e., we replace $\mathcal{R}_{a_1}$ and $\mathcal{R}_{a_3}$ by $P_{a_1}$ and $P_{a_3}$, respectively. This way, we obtained a grammar-based repair program.

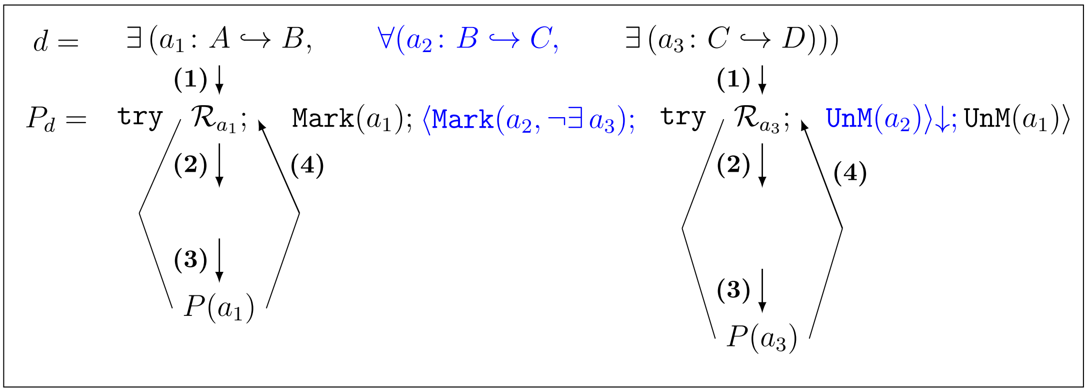

$d = \exists(a_1 \colon A \hookrightarrow B, \quad \forall(a_2 \colon B \hookrightarrow C, \quad \exists(a_3 \colon C \hookrightarrow D)))$
$P_d =$ try $\mathcal{R}_{a_1}$; Mark($a_1$); ⟨Mark($a_2, \neg\exists\, a_3$); try $\mathcal{R}_{a_3}$; UnM($a_2$)⟩↓; UnM($a_1$)⟩
(1)
(2)
(3)
(4)
$P(a_1)$
(1)
(2)
(3)
(4)
$P(a_3)$

Figure 4.16: Construction of an $GG$-based repair program [10]

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

[10] In the figure, Unmark is abbreviated by UnM.

**Definition 29 (compatibility).** Let $P_d$ be a repair program for a condition $d$. A set of rules $\mathcal{R}$ is *$d$-compatible* (w.r.t. $P_d$) if, for all rules in the repairing sets of $P_d$, there are equivalent transformations via $\mathcal{R}$. In particular, if $\mathcal{R} = \{\varrho\}$, we also say that $\varrho$ is $d$-compatible.

**Example 53.** The rule `AddTok` from Example 51 is compatible with the condition $c = (\ \boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{Tk}}\ )$.

**Remark.** Instead of equivalent transformations, one may consider approximating transformations: Two transformations $t, t'$ starting from the same graph are *replaceable* if for each extension of $t$, there is one for $t'$, and vice versa. A transformation $t'$ approximates $t$ w.r.t. ac, denoted by $t \leq_{\mathrm{ac}} t'$, if the transformations $t, t'$ are replaceable and, for all the triples $\langle g_1, h_1, i_1 \rangle \in [\![t_1]\!]$ and $\langle g_2, h_2, i_2 \rangle \in [\![t_2]\!]$, $h_1 \circ i_1 \models \mathrm{ac} \Leftrightarrow h_2 \circ i_2 \models \mathrm{ac}$.

If a rule set $\mathcal{R}$ is $d$-compatible w.r.t. the repair program $P$ for $d$, for all rules in the repair program, there are transformations via $\mathcal{R}$. These transformations via $\mathcal{R}$ can be transformed into $GG$-based programs. A transformation $t$ is equivalent to a program, denoted $t \equiv P(t)$, if for $t \colon g \Rightarrow h$, there exists a $t' \colon g \Rightarrow_{P(t)} h'$ via $P(t)$, such that $h' = h$.

**Theorem 6 (from transformations to grammar-based programs).** For every transformation $t \colon g \Rightarrow^*_{\mathcal{R}} h$, there is an $GG$-based program $P(t)$ such that $t \equiv P(t)$.

**Informal Description.** To get an $GG$-based program for a given transformation with a rule set $\mathcal{R}$, we intuitively do the following: for a one-step transformation $t \colon g \Rightarrow_{\varrho} h$ via a rule $\varrho$ with interface $X$, the result is another rule $\overline{\varrho}$. The left interface of the rule $\overline{\varrho}$ is equal to the domain of the input morphism $g$ of the transformation, and the right interface of the rule is equal to the domain of the result morphism $h$. The application condition is the shifted application condition of the rule $\varrho$, shifted to the left-hand side $G$ of the resulting rule. Furthermore, node-deleting rules are equipped with the dangling-edges operator. For arbitrary transformations, we inductively apply the construction along the transformation.

**Construction 17.** Let $t : g \Rightarrow^*_{\mathcal{R}} h$ be a transformation with $g \colon X \hookrightarrow G$ and $h \colon Y \hookrightarrow H$. For direct transformations $t \colon g \Rightarrow_{\varrho} h$ via a rule $\varrho = \langle x, p, \mathrm{ac}, y \rangle$ with interfaces $X$ and $Y$, let $P(t) := \langle \mathtt{Mark}(g' \circ x, \mathrm{ac}'); \overline{\varrho}'; \mathtt{Unmark}(h' \circ y) \rangle$ be the rule with left interface $g' \circ x$, $\overline{\varrho} = G \Rightarrow H$ be the rule $\varrho$ equipped with context, $\mathrm{ac}' = \mathrm{Shift}(g', \mathrm{ac})$ the left application condition for $\overline{\varrho}$, and $h' \circ y$ the right interface. For transformations $t \colon g = g_0 \Rightarrow^{n+1}_{\mathcal{R}} g_{n+1} = h$, with $t_1 \colon g_0 \Rightarrow^n_{\mathcal{R}} g_n$, and $t_2 \colon g_n \Rightarrow_{\varrho} g_{n+1}$, let $P(t) := \langle P(t_1); P(t_2) \rangle$.

**Example 54.** For the grammar $GG$ with the rule $\langle \texttt{AddTok}, \nexists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ \rangle$ and the transformation $t\colon \boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$ , the $GG$-based program is

$$P(t) = \langle \texttt{Mark}(\ \boxed{\text{Pl}}\ ); \langle\ \boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ , \nexists\ \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}\ \rangle; \texttt{Unmark}(\ \boxed{\text{Pl}}\ )\rangle.$$

The program intuitively marks a Pl-node, at that position a tok-edge is added. Afterwards, the Pl-node is unmarked.

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

A (graph) program consisting of rules of a graph grammar $GG$ is said to be a *$GG$-based program.*

**Corollary 1 (from grammar-based programs to transformations).** For every terminating $GG$-based program $P$ there is a transformation $t\colon g \Rightarrow^*_{\mathcal{R}} h$ such that $P(t) \equiv t$.

**Proof.** The Theorem is a consequence of Theorem 6 and Fact 16. □

If a rule set is $d$-compatible, there is a grammar-based repair program for $d$. The mapping repl replaces the repairing sets by equivalent $GG$-based programs.

**Theorem 7 (grammar-based repair).** Let $P_d$ be a repair program for a condition $d$. If $\mathcal{R}$ is $d$-compatible (w.r.t. $P_d$), then there is an $GG$-based repair program for $d$.

**Construction 18.** Let $P'_d = P_d[\text{repl}]$ where the mapping repl replaces the repairing sets by equivalent $GG$-based programs.

**Proof.** By assumption, for all rules in the repairing sets with interfaces $A$ of $P_d$, there are equivalent transformations via the rules of the graph grammar. By Theorem 6, the transformations can be transformed into equivalent $GG$-based repair programs. This yields a mapping repl which replaces the repairing sets with interfaces $A$ by equivalent $GG$-based programs with interface $A$. By the Leibniz's replacement principle, the repair programs $P_d$ and $P_d[\text{repl}]$ are equivalent. Thus, $P_d[\text{repl}]$ is an $GG$-based repair program for $d$. An illustration is given in Figure 4.17. □

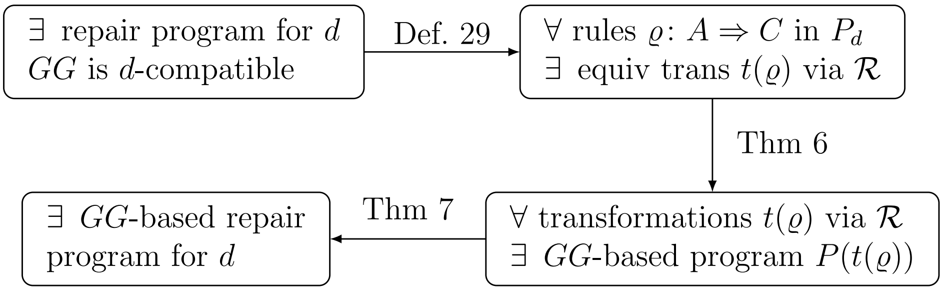

∃ repair program for $d$
$GG$ is $d$-compatible
Def. 29
∀ rules $\varrho\colon A \Rightarrow C$ in $P_d$
∃ equiv trans $t(\varrho)$ via $\mathcal{R}$
Thm 6
∃ $GG$-based repair program for $d$
Thm 7
∀ transformations $t(\varrho)$ via $\mathcal{R}$
∃ $GG$-based program $P(t(\varrho))$

Figure 4.17: Illustration of the proof for grammar-based repair programs

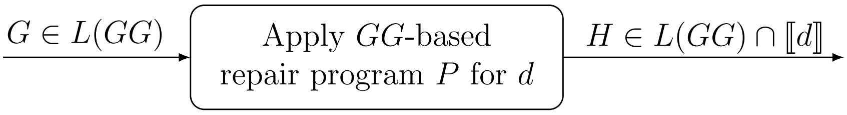
Whenever the given rule set $\mathcal{R}$ coincides with the rule set of the repair program for a constraint, then the $GG$-based repair program for the constraint coincides with the repair program for the constraint.

**Corollary 2 (grammar-based repair).** Let $\mathcal{R}$ be a rule set, $P_d$ be a repair program for $d$, $\mathcal{R}_d$ the repairing set of $P_d$, and $\mathcal{R} = \mathcal{R}_d$. Then the $GG$-based repair program $P_{\mathcal{R},d}$ for $d$ is equivalent the repair program $P_d$: $P_{\mathcal{R},d} \equiv P_d$.

**Proof.** The statement follows directly from Construction 18. □

In the following, we consider graph grammars as in Taentzer [Tae12]. Let $\mathcal{R}$ be the rule set of a graph grammar $GG$ and $P$ be an $GG$-based repair program for $d$. Then, for each transformation $G \Rightarrow_P H$ via the program, we have: If the graph $G$ belongs to the language of the grammar $GG$, then each resulting graph $H$ belongs to the language of the grammar $GG$ and satisfies the constraint $d$.

**Corollary 3 (grammar based repair).** Let $GG = (\mathcal{R}, S)$ be graph grammar and $P$ be an $GG$-based repair program for $d$. Then, for all transformations $G \Rightarrow_P \; H$ via $P$, $G \in L(GG)$ implies $H \in L(GG) \cap [\![d]\!]$.


$G \in L(GG)$
Apply $GG$-based repair program $P$ for $d$
$H \in L(GG) \cap [\![d]\!]$

**Proof.** The statement follows from Fact 16: Let $P$ be $GG$-based for $d$. Then for every transformation $G \Rightarrow_P H$, $H \models d$, and, since $P$ is $GG$-based, there exists $G \Rightarrow^*_{\mathcal{R}} H$, $H \models d$. By Fact 16, for every $G \in L(GG)$ there exists a transformation $G \Rightarrow^*_{\mathcal{R}} H$ such that $H \models d$. Thus, $H \in L(GG) \cap [\![d]\!]$. □

The compatibility problem turns out to be undecidable. This follows immediately from the undecidability of the coverability problem for increasing rule sets and positive conditions (see Bertrand et al. [BDK$^+$12]). The *coverability problem* is as follows. Given a graph grammar $GG = (\mathcal{R}, S)$ and a graph $C$, is there a graph $H$ such that $S \Rightarrow^*_{\mathcal{R}} H$ and $C \sqsubseteq H$, i.e., there is an injective morphism from $C$ to $H$.

**Lemma 27 (undecidability of compatibility).** The following problem is undecidable:

**Instance:** An increasing rule set $\mathcal{R}$ and a basic condition $\exists\,(A \hookrightarrow C)$.
**Question:** Is $\mathcal{R}$ $d$-compatible, i.e., $\exists\, t\colon A \Rightarrow^*_{\mathcal{R}} C'.\ C' \models \exists\,(A \hookrightarrow C)$?

**Proof.** The statement follows immediately from the undecidability of the coverability problem for increasing rule sets (Bertrand et al. [BDK$^+$12]). Assume the compatibility problem for increasing rule sets $\mathcal{R}$ and basic conditions $\exists\,(A \hookrightarrow C)$ is decidable. Then, the coverability problem for increasing rule sets $\mathcal{R}$, the start graph $A$ and final graph $C$ is decidable: By definition of compatibility and covering, $\mathcal{R}$ is $\exists\,(A \hookrightarrow C)$-compatible iff there is a transformation $A \Rightarrow^*_{\mathcal{R}} C'.C' {\models} \exists\,(A \hookrightarrow C)$ iff there is a transformation $A \Rightarrow^*_{\mathcal{R}} C'.C' \sqsupseteq C$, i.e., there is an injective morphism from $C$ to $C'$. Obviously, $C' \sqsupseteq C \Leftrightarrow C' \models \exists\,(A \hookrightarrow C)$, i.e., there is a covering. □

The compatibility problem is semi-decidable for arbitrary rule sets and arbitrary conditions. Let $\mathcal{R}$ be an arbitrary rule set and $d$ be a condition (with the corresponding repair program $P_d$ for $d$). For each repairing set in the repair program for $d$, we construct the set of all transformations which are equivalent to a transformation with the rule in the repairing set. If this yields a graph satisfying the condition, the algorithm terminates and returns true. In that case, the rule set is $d$-compatible. In general, this construction may not terminate. If the termination is requested, the construction aborts and cannot decide if there is an equivalent transformation. Consequently, it is semi-decidable.

**Lemma 28 (semi-decidability of compatibility).** For every finite rule set $\mathcal{R}$ and every condition $d$, $\mathcal{R}$-compatibility of $d$ is semi-decidable.

**Proof.** The algorithm in Figure 4.18 returns true, provided $\mathcal{R}$ is $d$-compatible or terminates with an exception, i.e., it is semi-decidable. □

**Informal Description.** The algorithm in Figure 4.18 constructs for each repairing set $\mathcal{R}_a$ or $\mathcal{S}_a$ in the condition $d$ an equivalent transformation, which is added to the set of equivalent transformations. This is done, up to a certain length. If we have found a graph, satisfying the constraint, we have found an equivalent transformation. If this is not the case, we may increase the length of transformation. If the termination is requested, we cannot decide, because there may be a transformation of a larger length.

```
Input: rule set $\mathcal{R}$, legit condition $d$, $maxRound \in \mathbb{N} \cup \{\infty\}$
Output: true, $\mathcal{R}$ is $d$-compatible throws UndecidedException;
$\mathcal{T}(a, 0) \leftarrow \emptyset, \quad i, j \leftarrow 1$ ;                  // initialize
/* $\mathcal{R}_a(d)$ ($\mathcal{S}_a(d)$) repairing sets in $d$ behind (non)-existential
   quantifier.                                                              */
for $\varrho \in \mathcal{R}_a(d)$ do
    repeat
        /* Construct set of equivalent transformations $A \Rightarrow C'$ of length $i$
           */
        $t(a, i) \leftarrow \text{constructTrafo}(a, \mathcal{R}, i)$;
        $\mathcal{T}(a, i) \leftarrow \mathcal{T}(a, i - 1) \cup t(a, i)$;
        $i \leftarrow i + 1$;
        if $i \geq maxRound$ then
            return throws UndecidedException;
            /* If termination is requested, we cannot decide   */
        end
    until $C' \models \exists\, a$;
    /* until an equivalent transformation is found          */
end
for $\varrho \in \mathcal{S}_a(d)$ do
    repeat
        /* ....                                                  */
    until $A' \models \nexists\, a$;
end
return true;
/* for all $\varrho \in \mathcal{R}_a(d), \mathcal{S}_a(d)$ an equivalent transformation is
   found                                                                    */
```

Figure 4.18: Algorithm for semi-decidability of compatibility in pseudocode

## 4.8 Related work

In this section, we present some related concepts on graph repair. The most significant one is the one of Schneider et al. [SLO19]. We compare our approach with the one of Schneider et al. in detail: first, we give an introduction to the approach of Schneider, then, we show that repair programs induce (graph) repairs, afterwards, we compare the termination of the approaches and compare the notion of maximal preservation and “least-changing” graph repairs, i.e., we conjecture, that solid repair programs induce “least-changing” graph repairs. Another approach is the one of Cheng et al. [CCYW18]. The related work to model repair is presented at the end of the next chapter.

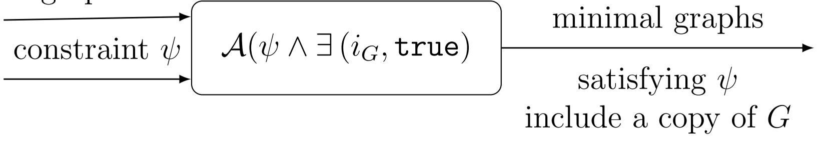
### Approach of Cheng et al.

In **Cheng et al.** [CCYW18], a rule-based approach for graph repair is presented. Given a set of rules, and a graph, they use this set of rules, to handle different kinds of conditions, i.e., incompleteness, conflicts and redundancies. The rules are based on seven different operations not defined in the framework of the DPO-approach. They look for the “best” repair based on the “graph edit distance”.

### Approach of Schneider et al.

In **Schneider et al.** [SLO19], a logic-based incremental approach to graph repair is presented, generating a sound and complete (upon termination) overview of least changing repairs. They formalize consistency by graph conditions being equivalent to first-order graphs formulas in the sense of Courcelle [Cou97]. They present several repair algorithms: Two state-based repair algorithms which restore consistency independent of the graph update history, whereas delta-based (or incremental) repair algorithms take the history explicitly into account.

The algorithms rely on an existing model generation algorithm for graph conditions implemented in `AutoGraph` [SLO18]. The tool `AutoGraph` determines the operation $\mathcal{A}$ that constructs a finite set of all minimal graphs satisfying a given constraint $\psi$.



The first state-based graph repair algorithm takes a graph and a graph constraint as inputs and returns a set of graph repairs. Given a condition and a graph, they compute a set of symbolic models, which cover the semantics of a graph condition. The first state-based algorithm is sound, i.e., all results are least changing repairs, and complete (upon termination) with respect to non-deleting repairs.

The second state-based algorithm computes all least changing repairs. The algorithm uses the approach of the first state-based algorithm, but computes $\mathcal{A}(\psi \wedge \exists\,(i_G, \texttt{true}))$ whenever an inclusion $l\colon G_c \hookrightarrow G$ describes how $G$ can be restricted to one of its subgraphs $G_c$. Every graph $G'$ obtained from the application of $\mathcal{A}$ for some of these graphs $G_c$ then results in one graph repair returned by the algorithm. The algorithm is sound and complete whenever the call to `Autograph` terminates.

For the delta-based approach, the repair algorithms may contain, in addition to the graph as in the state-based algorithms, an additional satisfaction tree (ST) for encoding if and how a graph satisfies a graph condition.

All approaches are proven to be **correct**, i.e., the repair programs yield a graph satisfying the condition. The delta-based repair algorithm takes the graph update history explicitly into account, i.e., the approach is **dynamic**. In contrast, our approach is static, i.e., we first construct a repair program, then apply this program to an arbitrary graph. The repair algorithm does not **terminate** if the repair updates trigger each other ad infinitum. Here, we have constructed terminating repair programs.

## Similarities and differences

In Schneider et al. [SLO19], the notions of update, graph repair, and least changing graph repair are introduced.

An update of a graph $G$ resulting in a graph $H$ can be represented by a pair of injective morphisms with the same domain, denoted by $\langle G \hookleftarrow D \hookrightarrow H\rangle$. The domain $D$ of the morphisms represents the part of $G$ that is preserved by the update. An update $u_1$ is a sub-update of $u$ whenever the modifications defined by $u_1$ are fully contained in the modifications defined by $u$. Intuitively, this is the case when $u_1$ can be composed with another update $u_2$ such that the resulting update has the same effect as $u$ and $u_2$ does not delete any element that was added before by $u_1$. This is stated by requiring that $D$ is the intersection (pullback) and that the graph $H$ is its union (pushout), i.e., by requiring that the diagram (1) in Figure4.19 is a pullback and a pushout.

**Definition 30 (update & sub-update [SLO19]).** An *update* $\langle G \hookleftarrow D \hookrightarrow H\rangle$ is a pair of injective morphisms. An update $u_1 = \langle G \hookleftarrow D_1 \hookrightarrow H\rangle$ is a *sub-update* of $u = \langle G \hookleftarrow D \hookrightarrow M\rangle$, written $u_1 \leq u$, if there is an update $u_2 = \langle H \hookleftarrow D_2 \hookrightarrow M\rangle$ such that diagram (1) in Figure 4.19 is a pullback and pushout and the triangles commute. Moreover, we write $u_1 < u$ if $u_1 \leq u$ and not $u \leq u_1$. For an update $\langle G \hookleftarrow D \hookrightarrow H \rangle$ with $G \cong D$, we shortly write $G \Rightarrow H$.

$$\begin{tikzcd}
G & & D_1 \arrow[ll, hook', "l_1"'] \arrow[rr, hook, "r_1"] \arrow[d, phantom, "="] & & H & & D_2 \arrow[ll, hook', "l_2"'] \arrow[rr, hook, "r_2"] \arrow[d, phantom, "="] & & M \\
& & & & (1) & & & & \\
& & & & D \arrow[llllu, hook] \arrow[lluu, hook] \arrow[rruu, hook] \arrow[rrrruu, hook] & & & &
\end{tikzcd}$$

Figure 4.19: A sub-update of an update

**Example 55.** Given the constraint $d = \nexists( \boxed{\text{Pl}} \overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}} )$ meaning there do not exist two tok-edges between a Pl-node and a Tk-node, and the graph $G = \boxed{\text{PN}} \xrightarrow{\text{place}} \boxed{\text{Pl}} \overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}}$ , there is an update

$$u = \boxed{\text{PN}} \xrightarrow{\text{place}} \boxed{\text{Pl}} \overset{\text{tok}}{\underset{\text{tok}}{\rightrightarrows}} \boxed{\text{Tk}} \Rightarrow \boxed{\text{PN}} \xrightarrow{\text{place}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}} .$$

It is also canonical, since the morphism $r$ is an isomorphism. The update $u$ is a graph repair for $G$: the graph $H = \boxed{\text{PN}} \xrightarrow{\text{place}} \boxed{\text{Pl}} \xrightarrow{\text{tok}} \boxed{\text{Tk}}$ satisfies the constraint $d$.

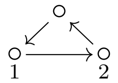
Given a constraint, an update is a (graph) repair if the resulting graph satisfies the constraint. It is least changing if it cannot be decomposed into repairing sub-updates.

**Definition 31 (repair & least changing repair [SLO19]).** Given a constraint $d$, an update $u = \langle G \hookleftarrow D \hookrightarrow H \rangle$ is a *(graph) repair* for $G$, if $H$ satisfies $d$. A repair $u$ for $G$ and $d$ is *least changing* if there is no repair $u'$ for $G$ and $d$ such that $u' < u$. The class of all (least changing) repairs of $G$ with respect to $d$ is denoted by $\mathcal{U}(G, d)$ ($\mathcal{U}_{\text{lc}}(G, d)$).

**Example 56.** Consider the graph $G = \circ \longrightarrow \circ$ and the constraint $d = \forall( \underset{1}{\circ} , \exists\ \underset{1}{\circ} \longrightarrow \circ )$ meaning that for each node there is a real outgoing edge. By Definition 31, the updates below are repairs for $G$ with respect to $d$.

$$u_1 \quad = \quad \underset{1}{\circ} \longrightarrow \underset{2}{\circ} \quad \Rightarrow \quad \underset{1}{\circ} \rightleftarrows \underset{2}{\circ}$$

$$u \quad = \quad \underset{1}{\circ} \longrightarrow \underset{2}{\circ} \quad \Rightarrow$$



There is an infinite set of repairs for $G$ with respect to $d$, e.g., those containing a directed cycle on the right-hand side. The repairs are not in $\leq$ relation: $u_1 \not\leq u$

Choosing $D_2 \cong H$, diagram (1) becomes a pushout and a pullback, but there is no injective morphism $D_2 \hookrightarrow M$. If we take a graph $D_2$ with morphisms to $H$ and $M$ as shown in Figure 4.20, then the diagram (1) does not become a pushout. Analogously, one can show that $u \not\lesssim u_1$. The repairs $u_1$ and $u$ are least changing.

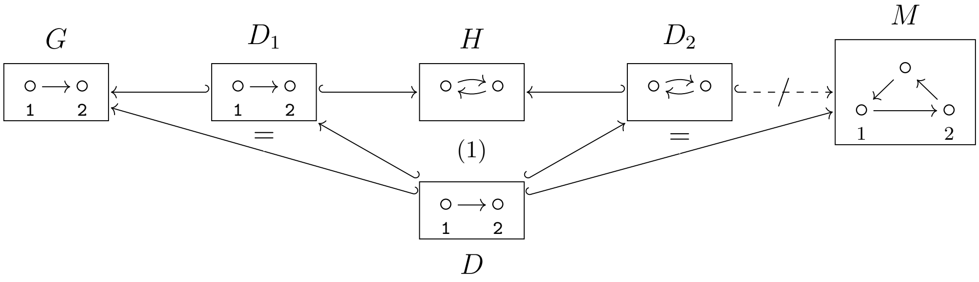


Figure 4.20: The update $u_1$ is no sub-update of $u$

There is a close relationship between the repair programs and the graph repair of Schneider et al.: By the definitions of repair programs and repairs, repair programs induce repairs. Maximally preserving repair programs induce *maximally-preserving* repairs.

**Corollary 4 (repair programs induce repairs).** Let $G$ be a graph, $d$ be a constraint, and $P$ be a repair program for $d$. Then $P$ induces a non-empty set of repairs for $G$ with respect to $d$.

$$\mathcal{U}(G, P) = \{\langle G \hookleftarrow \operatorname{dom}(\operatorname{tr}(t)) \hookrightarrow H\rangle \mid t\colon G \Rightarrow_P H\}$$

where $\operatorname{tr}(t)$ denotes the partial track morphism $\operatorname{tr}(t)\colon G \rightharpoonup H$ (see, e.g., [Plu05]).

**Proof.** By the existence requirement in the definition of repair programs, there exists transformation $G \Rightarrow_{P,i} H$. By the correctness requirement in the definition of repair programs, $u = \langle G \hookleftarrow \operatorname{dom}(\operatorname{tr}(t)) \hookrightarrow H\rangle \in \mathcal{U}(G, P)\,.\, H \models d$, i.e., $u$ is a repair for $G$ with respect to $d$. □

We conjecture that all repairs induced by the solid repair program $P_d$ for a constraint $d$ are least changing.

**Conjecture (solid repair programs induce least changing repairs).** Let $G$ be a graph, $d$ be a constraint, and $P_d$ be a solid repair program for $d$. Then $P_d$ induces a non-empty set of least changing repairs for $G$ with respect to $d$, i.e., $\mathcal{U}(G, P_d) \subset \mathcal{U}_{\mathrm{lc}}(G, d)$.

The solid construction gets a legit condition as input and returns a maximally preserving repair program. The state-based algorithms of Schneider et al. get an arbitrary constraint as input and return a set of least changing repairs. All approaches are proven to be sound, i.e., every repair of a graph w.r.t. a constraint is least changing (maximally preserving). The first state-based algorithm ($\mathcal{R}\text{epair}_{sb_1}$) is complete (upon termination) with respect to non-deleting, least changing repairs. The second state-based algorithm ($\mathcal{R}\text{epair}_{sb_2}$) is complete (upon termination) with respect to least changing repairs. There is a finite set of maximally preserving repairs up to isomorphism for a graph and a constraint. The solid repair programs induce the complete set of maximally preserving repairs. Consequently, we think, that the notion of maximally preserving repair programs is a more suitable notion than the least-changing repairs. The algorithms of Schneider et al. do not terminate, the solid repair programs in this thesis are terminating for a subclass of all conditions. The results are summarized in Table 4.1.

| | condition | property | soundness | completeness | termination |
|---|---|---|---|---|---|
| $\mathcal{R}\text{epair}_{sb_1}$ | arbitrary constraint | least changing | + | -$^*$ | - |
| $\mathcal{R}\text{epair}_{sb_2}$ | arbitrary constraint | least changing | + | (+)$^\sharp$ | - |
| solid construction | legit condition | maximally preserving | + | + | + |

where $^*$ w.r.t non-deleting repairs, $^\sharp$ complete (upon termination)

Table 4.1: Comparison with the approach of Schneider et al.

(1) **Input.** The main difference is that Schneider et al. allow arbitrary constraints as input and may get non-termination while in this thesis, we restrict on constraints (conditions) with repair program guaranteeing termination. However, it is not clear, under which prerequisites the approach of Schneider et al. terminates.

(2) **Least changing & maximally preserving.** Example 56 shows that a maximally preserving repair program yields one maximally-preserving repair while there is an infinite number of least changing repairs. The notion of least changing repair seems to be coarse. The least changing repair $u$ in Example 56 is not induced by a solid repair program.

**Remark.** It would be important to investigate how the conditions, for which the algorithms of Schneider et al. terminate, are related with legit conditions.

**Bibliographic notes.** Graph repair was first investigated by [HS18]. The notion of a repair program is defined first in [HS18, SH19, San20]. For constraints, the notions update and repair and defined first in Schneider et al. [SLO19].

## 4.9 Conclusion

In this chapter, we have presented the theory of typed repair programs. Application of the typed repair programs to an arbitrary typed graph yields a typed graph satisfying the condition. The repair programs are derived directly from the given condition.

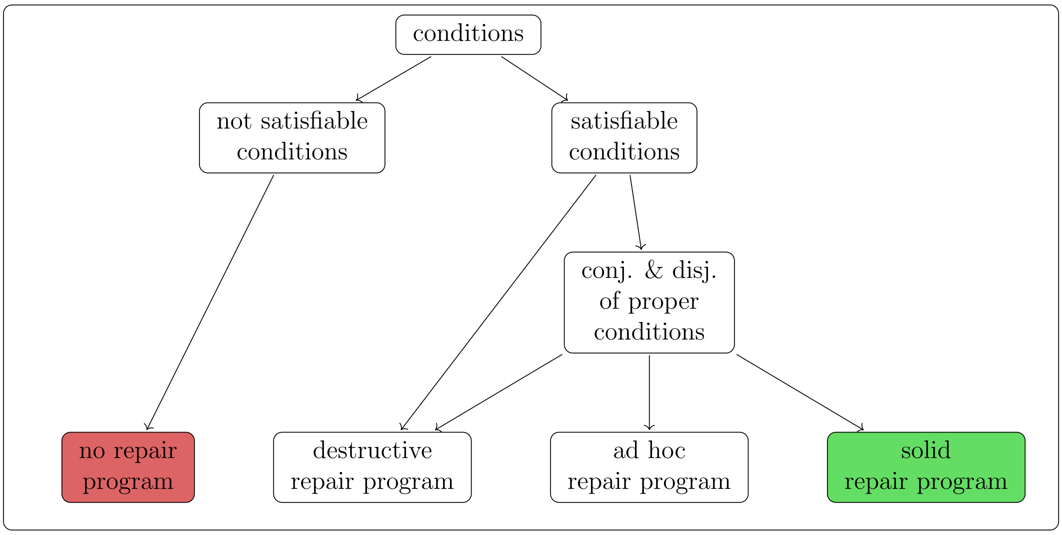


Figure 4.21: Overview of different repair programs for conditions

Figure 4.21 provides an overview of the conditions and the corresponding repair programs. For conditions, which are not satisfiable, there cannot exist a repair program. For satisfiable conditions, there exists a destructive repair program, which deletes an input graph and creates a new graph satisfying the condition. For conjunctions and disjunctions of proper conditions there exists the destructive repair program, the ad hoc repair program and the solid repair program. The ad hoc repair programs are stable. The solid repair programs were constructed to be stable, maximally preserving, and terminating. Additionally, we have considered grammar-based repair where the repair programs are constructed from a given set of rules and a given condition. Based on the repair program for legit conditions, we have constructed a grammar-based repair program for legit conditions, provided that the given rule set is compatible with the repairing sets of the original program.

Summarizing, we have constructed typed

1. **Destructive repair programs** for all satisfiable conditions (Lemma 16).
2. **Solid & ad hoc repair programs** for a large class of satisfiable conditions: proper conditions, preserving conjunctions and disjunctions of proper conditions (Theorems 2, 3, 4, and 5). The program properties are summarized in Table 4.2.

| repair program | stable | max pres | terminating |
|---|---|---|---|
| destructive | - | - | - |
| ad hoc | + | - | - |
| solid | + | + | + |

Table 4.2: Program properties for repair programs

3. **Grammar-based repair programs** may be seen as a refinement of the programs above, which guarantee that whenever the input belongs to the language of a grammar, the result is in the language of a given grammar.

# Chapter 5

# Application to meta-modeling

In this chapter, we apply the theory on typed graph repair to meta-modeling.

To enable automated model repair or model completion, we look for an algorithm that - given a meta-model with two constraints and any model satisfying one of the constraints - creates another model satisfying the old as well as a new one (see Figure 5.1).

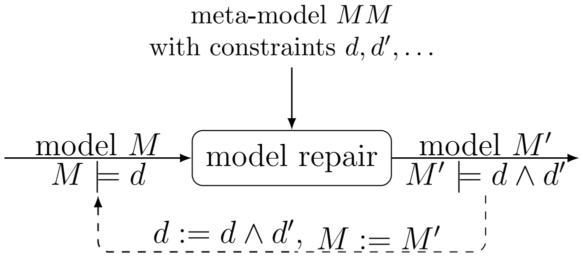


Figure 5.1: General idea to model repair

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

A model graph based on the Eclipse Modeling Framework (EMF) [SBMP08], short EMF-model graph, is a typed graph satisfying some structural EMF constraints. Application of the results for typed graphs to the EMF world yields model repair and completions, i.e., the resulting typed graph satisfies all EMF constraints. To do so, we construct model repair and completion programs for EMF$k$ constraints, a first-order version of the EMF constraints, where a natural number $k$ restricts the constraints to that size $k$, such that the application to a typed graph yields an EMF$k$ -model graph. The results known from typed graph repair are applied to EMF$k$ -model repair and EMF-model repair.

In Section 5.1, we introduce meta-models. In Section 5.2, we introduce EMF-model graphs, represent EMF$k$ constraints as graph conditions, and show that the constraints on so-called "opposite edges" are preserving. In Section 5.3, we show that there are model repair and model completion programs for the EMF$k$ constraints, and EMF constraints. In Section 5.4, we present some related concepts on model repair.

## 5.1 Meta-modeling

The following introduction is oriented at Heckel and Taentzer [HT20], Jeusfeld [Jeu18], and the Object Management Group (OMG) Model Driven Architecture (MDA) Guide [Gro03], and the thesis by Rutle [Rut10].

Models are used in software development as abstract representations of systems. Models may be represented as graphical or textual forms, depending on their purpose. A textual representation is the Object Constraint Language, while class diagrams based on the Unified Modeling Language (UML) are visual. In the OMG specification, a meta-model is defined as a "model of models, where a *model* of a system is a description or specification of that system and its environment for some certain purpose". We give a more precise definition from Jeusfeld [Jeu18].

A *meta-model* is a model that consists of statements about models. Hence, a meta-model is also a model but its universe of discourse is a set of models, namely, those models that are of interest to the creator of the meta-model. The statements in a meta-model can define the constructs or can express true and desired properties of the constructs. Like models are abstractions of some reality, meta-models are abstractions of models. The continuation of the abstraction leads to meta-meta-models, being models of meta-models containing statements about meta-models (see Figure 5.2).

This means, that a model at a certain level of abstraction conforms to a meta-model at the abstraction level above, and acts as a meta-model for models at the level below. Theoretically, this may continue ad infinitum. According to the OMG's vision of MDE,. models, modeling languages, and meta-modeling languagues are oganized in four levels, $M_0 - M_3$, in the so-called OMG's 4-layered hierarchy [BG01] . The most agreed interpretation of the OMG's four layered hierarchy is summarized as follows (see Figure 5.2)

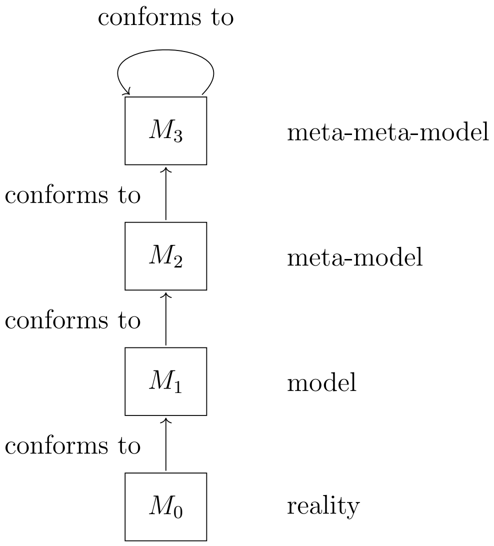


Figure 5.2: Illustration of (meta)-modeling languages and their corresponding meta-models

- Level $M_0$ contains the reality (e.g., in this thesis, the Petri-net world)
- Level $M_1$ contains models (in this thesis, formalized as typed graphs, see Example 3)
- Level $M_2$ contains meta-models (in this thesis, formalized as type graphs (with containment), see Example 2)
- Level $M_3$ contains meta-meta-models.

**Remark.** In newer papers, only the levels $M_1$ to $M_3$ are considered. While the upper two "conforms to" relations are "instance of" relations, the lower one is a "models" relation.

## 5.2 EMF-model graphs

The standard tool for model-driven engineering is the Eclipse Modeling Framework (EMF). In [BET12], an EMF-model graph is defined as a typed graph, representing the model, satisfying the following conditions: No node has more than one container. There are no two parallel edges of the same type. No cycles of containment occur. For all edges in the opposite edges relation, there exists an edge in opposite direction.

**Definition 32 (EMF-model graph).** Given a type graph $TG$, a typed graph $G$ is an *EMF-model graph*, if it satisfies the following conditions:

1. At most one container: $\forall e_1, e_2 \in C_G.\ \mathrm{t}_G(e_1) = \mathrm{t}_G(e_2)$ implies $e_1 = e_2$.
2. No containment cycle: $\forall v \in V_G.\ (v, v) \notin \mathtt{cont}_G$.
3. No parallel edges: $\forall e_1, e_2 \in E_G.\ \mathrm{s}_G(e_1) = \mathrm{s}_G(e_2)$, $\mathrm{t}_G(e_1) = \mathrm{t}_G(e_2)$, and $\mathit{type}_{E_G}(e_1) = \mathit{type}_{E_G}(e_2)$ implies $e_1 = e_2$.
4. All opposite edges: $\forall (e_1, e_2) \in O.\ \forall e'_1 \in E_G.\mathit{type}_G(e'_1) = e_1.\ \exists e'_2 \in E_G.\ \mathit{type}_G(e'_2) = e_2, \mathrm{s}_G(e'_1) = \mathrm{t}_G(e'_2)$ and $\mathrm{s}_G(e'_2) = \mathrm{t}_G(e'_1)$.

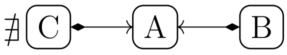
The set $C_G$ denotes the set of edges in $G$ which are typed by a containment edge. $\mathtt{cont}_G \subseteq V_G \times V_G$ is the *containment relation* induced by the set $C \subseteq E_T$: If $e \in C$ and $v_1 \leq \mathrm{s}(e), v_2 \leq \mathrm{t}(e)$, then $(v_1, v_2) \in \mathtt{cont}_G$.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

**Remark.** The EMF-model graphs in Biermann et al. [BET12] also contains inheritance. It would be important to extend the approach to typed graphs with containment and inheritance (see further topics)

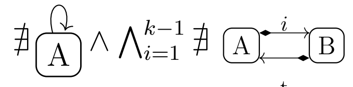
The second constraint is a monadic second-order constraint. Instead of it, we consider the constraint "No containment cycle of length $\leq k$" for a fixed natural number $k$. The resulting constraints, called *EMFk constraints*, are first-order constraints and can be expressed by typed graph constraints [HP09].

**Fact 17 (EMF$k$ constraints).** For the EMF$k$ constraints, there is a schema of typed graph constraints:

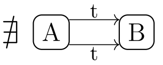
1. At most one container



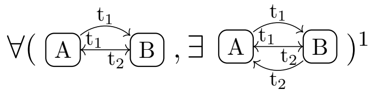
2. No containment cycle of length $\leq k$



3. No parallel edges



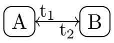
4. All opposite edges



where A,B,C,t,$\mathrm{t}_1$,$\mathrm{t}_2$ are node and edge types, respectively, edges without type are arbitrary typed, and $\xrightarrow{i}$ denotes a path of containment edges of length $i$.

[1] The bidirectional edge indicates, that the edges are in the opposite-edge relation. The presentation slightly differs from the presentation in the literature (e.g., Nassar [Nas20], Wang [Wan16]) (see remark below).

The first constraint requires that there are no two different containment edges with a common target. The second constraint requires that there are no loops and no cycles of length $\leq k$. The third constraint requires that there are no parallel edges of the same type. The fourth constraint requires that, if there is an opposite-edge marking between an A-typed and a B-typed node with type requirement $\mathrm{t}_1$,$\mathrm{t}_2$, there exists already one edge $e_1$ with the type $\mathrm{t}_1$, then an opposite edge with type $\mathrm{t}_2$ in opposite direction should exist.

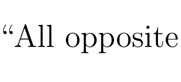
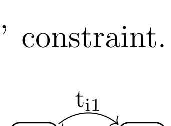
**Remark.** The presentation of the fourth constraint slightly differs from the presentation in the literature (Nassar [Nas20], Wang [Wan16]). In graphs, opposite edges are not represented explicitly. A type graph (with containment) consists of a graph with a distinguished set of containment edges and a relation of opposite edges. Since a type graph is not a graph, we have integrated all information on opposite edges and containment edges into the type graph, i.e., the edges in the opposite edges relation are represented by bidirectional edges and containment edges are marked with a black diamond. On the instance level, we cannot decide whether an edge is in the opposite edge relation or not. Consequently, we add a bidirectional edge between the A-node and the B-node if the edge is in the opposite edge relation, i.e.,  into the typed graph. If we would not add this information, we could not distinguish between the edges in the opposite edge relation and those which are not.

**Fact 18.** The instances of EMF$k$ constraints are negative or universal. Every conjunction of EMF$k$ constraints instances is satisfiable.

**Lemma 29 (preservation of EMF$k$ constraints).** Every conjunction of EMF$k$ constraint instances is preserving.

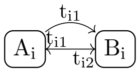
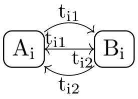
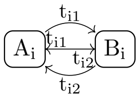
**Proof.** The instances of the first three EMF$k$ constraints are negative; thus, the sequences are preserving. The instances of the forth EMF$k$ constraint are universal; by induction on the number of constraints, we show that the sequences are preserving.

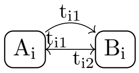
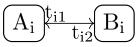
Let $d_i$ be an instance of the "All opposite edge" constraint.

$d_i = \forall($  $, \exists$  $)$

The "All opposite edge" requirement in $d_i$ is abbreviated by $\langle A_i, \mathrm{B}_i, \mathrm{t}_{i1}, \mathrm{t}_{i2}\rangle$. The edges in $d_i$ are called $\langle \mathrm{A}_i, \mathrm{B}_i, \mathrm{t}_{i1}\rangle$ and $\langle \mathrm{B}_i, \mathrm{A}_i, \mathrm{t}_{i2}\rangle$-edges, respectively.

The repair program for $d_i$ according to Theorem 3 is $P'_i = \langle P_i; P^{\text{id}}_{\nexists a,i}\rangle$, with

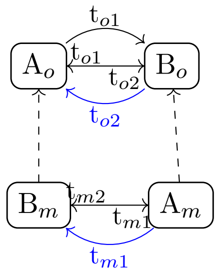
$P_i =$ $\langle \texttt{Mark}(a, \neg c);$  $\Rightarrow$  $, \nexists$  $; \texttt{Unmark}(a)\rangle\downarrow.$

$P^{\text{id}}_{\nexists a,i} =$ $\langle \texttt{Mark}(a, \neg c);$  $\Rightarrow$  $\rangle\downarrow$

Let $n = 1$ and $G \Rightarrow_{P'_1} H$ with $G \models \texttt{true}$. Then $P'_1$ is $\texttt{true}$-preserving.

Let $n \to n + 1$ and $G \Rightarrow_{\langle P'_1;\ldots;P'_{n+1}\rangle} M$ be an arbitrary transformation. Then the transformation is of the form $G \Rightarrow_{\langle P'_1;\ldots;P'_n\rangle} H \Rightarrow_{P'_{n+1}} M$. By induction hypothesis, $P'_1, \ldots, P'_n$ is $d_0, \ldots, d_n$-preserving, i.e., for $k = 1, \ldots, n$, $P_k$ is $\wedge^n_{i=0} d_i$-preserving.

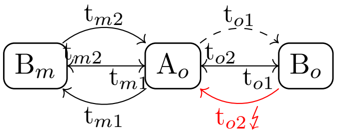
Consider now the transformation step $H \Rightarrow_{P'_{n+1}} M$. Then $M$ consists of edges from $H$ as well as edges created by the program $P_{n+1}$, or of the edges from $H$ and the edge $\langle \text{A}_{n+1}, \text{B}_{n+1}, \text{t}_{n+1,1}\rangle$ deleted by the program $P^{\text{id}}_{\nexists a,n+1}$. If we can show that $P'_1, \ldots, P'_n$ does not change the graph $M$, then the graph $M \models \wedge^n_{i=1} d_i$, and, since $P'_{n+1}$ is a repair program for $d_{n+1}$, $M \models d_{n+1}$. This holds for arbitrary transformations. In the following, let $o := n + 1$.



Consider a $\langle \text{B}_o, \text{A}_o, \text{t}_{o2}\rangle$-edge created in the last transformation step. Then there exist a $\langle \text{A}_o, \text{B}_o, \text{t}_{o1}\rangle$-edge in $M$ with opposite-edge requirement
$\langle \text{A}_o, \text{B}_o, \text{t}_{o1}, \text{t}_{o2}\rangle$. If there is a match for the plain rule of the program $P_m$ $(m < o)$ using the created edge, then a $\text{A}_m = \text{B}_o$, $\text{B}_m = \text{A}_o$, $\text{t}_{m1} = \text{t}_{o2}$.

Assume $t_{o2} = t_{m1}$, i.e., the created edge has the same type as another edge in the typed graph, with the same source and target node. By the definition of type graphs (Definition 2), the relation $O$ is functional, i.e., there is only one opposite-edge requirement between the nodes in consideration. Consequently, $\text{t}_{m2} = \text{t}_{o1}$. Thus, there is an opposite-edge for the edge in consideration, i.e., the application condition of the rule in $P_m$ is not satisfied and $P_m$ cannot be applied. Thus, $P_1 \ldots, P_{n+1}$ is $d_0, \ldots, d_{n+1}$-preserving.

It remains to show that the decreasing program $P^{\text{id}}_{\nexists a,i}$ is not applicable. Consider a $\langle \text{A}_o, \text{B}_o, \text{t}_{o1}\rangle$-edge deleted in the last transformation step. Then there exist a $\langle \text{B}_o, \text{A}_o, \text{t}_{o2}\rangle$-edge in $M$ with opposite-edge requirement, but the $\langle \text{B}_o, \text{A}_o, \text{t}_{o2}\rangle$-edge cannot be added because it would introduce a new violation of the "At most one container" or "No containment cycle of length $\leq k$" constraints (see below).



Assume $\mathrm{t}_{o1} = \mathrm{t}_{mi}$, i.e., the deleted has the same type as another edge in the typed graph, with the same source and target node. By the definition of type graphs (Definition 2), the relation $O$ is opposite direction, i.e., $\forall(e_1, e_2) \in O$, $\mathrm{s}(e_1) = \mathrm{t}(e_2)$ and $\mathrm{s}(e_2) = \mathrm{t}(e_1)$. Consequently, for each edge with type $\mathrm{t}_{m1}$, there exists an edge $\mathrm{t}_{m2}$ and for all $m$, $(m < o)$, the application condition $\neg c$ of the program $P^{\mathrm{id}}_{\nexists a,m}$ is not satisfied, and $P^{\mathrm{id}}_{\nexists a,m}$ cannot be applied and the opposite edge pair is repaired. Thus, $P^{\mathrm{id}}_{\nexists a,1}. \dots, P^{\mathrm{id}}_{\nexists a,n+1}$ is $d_0, \dots, d_{n+1}$-preserving. Since $P_1. \dots, P_{n+1}$ and $P^{\mathrm{id}}_{\nexists a,1}. \dots, P^{\mathrm{id}}_{\nexists a,n+1}$ are $d_0, \dots, d_{n+1}$-preserving, it follows that $P'_1. \dots, P'_{n+1}$ is $d_0, \dots, d_{n+1}$-preserving.

This completes the inductive proof. □

## 5.3 EMF-model repair

Let $\mathrm{emf}k_1, \mathrm{emf}k_2$ and $\mathrm{emf}_1, \mathrm{emf}_2$ be conjunctions of EMF$k$ and EMF constraints, respectively. An *EMFk -model repair* program for $\langle \mathrm{emf}k_1, \mathrm{emf}k_2 \rangle$ is an $\mathrm{emf}k_1$-preserving repair program for $\mathrm{emf}k_2$. An *EMF-model repair* program for a typed graph $L \models \mathrm{emf}_1$ and $\mathrm{emf}_2$ is a program $P$ such that, for all transformations $L \Rightarrow_P M$, $M \models \mathrm{emf}_1 \wedge \mathrm{emf}_2$. An *EMFk -model completion* program is an EMF-model repair program for `true` and the conjunction of all EMF$k$ -constraints. An *EMF-model completion* for a typed graph $L$ is a program $P$ such that, for all transformations $L \Rightarrow_P M$, $M$ is an EMF-model graph. A model repair or completion program $P$ for a constraint $e$ is *stable* if, for all transformations $L \Rightarrow_P M$, and all typed graphs $L \models e$ implies $L \cong M$, i.e., they do not change the typed graph, provided the constraint $e$ is satisfied.

**Theorem 8 (EMF$k$ -model repair & completion).** Let emf$k_1$ be a conjunction of negative EMF$k$ constraints, emf$k_2$ a conjunction of negative or universal and preserving EMF$k$ constraints, and $\overline{\text{emf}k}$ the conjunction of all EMF$k$ constraints.

1. There is a model repair program for $\langle \text{emf}k_1, \text{emf}k_2 \rangle$.
2. There is a model completion program.
3. The EMF$k$ -model repair and completion programs are stable.

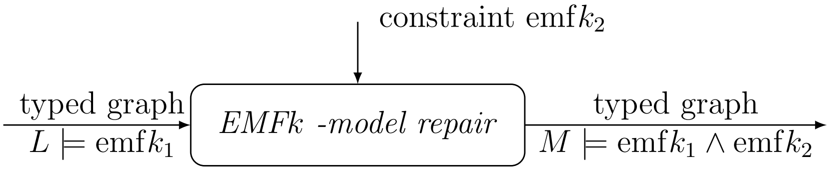

constraint emf$k_2$
typed graph
$L \models \text{emf}k_1$
*EMFk -model repair*
typed graph
$M \models \text{emf}k_1 \wedge \text{emf}k_2$

Figure 5.3: Illustration of EMF$k$ -model repair

**Proof.** 1. By Theorem 2, there are repair program $Q_1, Q_2$ for emf$k_1$, emf$k_2$, respectively. By Lemma 29, every conjunction emf$k_1$ and emf$k_2$ is preserving. Consequently, Lemma 20 can be applied and there is an emf$k_1$-preserving repair program ${Q'_2}^{\text{emf}k_1}$ for emf$k_2$, and by Fact 13, $\langle Q_1, {Q'_2}^{\text{emf}k_1} \rangle$ is a repair program for emf$k_1 \wedge$ emf$k_2$. 2. The statement is an immediate consequence of Theorem 8.1. 3. The statement follows immediately from Construction 13. □

**Remark ((OCL-)Constraints).** By the repair results on typed graphs, (model) repair and completion can be done for other constraints satisfying the requirements in Theorem 3, e.g., for first-order (OCL-)constraints.

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

**Fact 19.** The application of the EMF$k$ -model repair (completion) program does not change the number of nodes.

There is a close relationship between EMF$k$ and EMF (see Figure 5.4). For an EMF$k$ constraint emf$k$, emf denotes the more rigorous EMF constraint requiring no containment cycles and, for an EMF constraint emf, emf$k$ denotes the weaker EMF$k$ constraint emf requiring no containment cycles of length $\leq k$.

$$\begin{tikzcd}
\mathrm{emf}_1 \Dashv L \arrow[r, Rightarrow, "P"] \arrow[d, Rightarrow, "\text{if } |V_L| \leq k"'] & M \models \mathrm{emf}_1 \wedge \mathrm{emf}_2 \\
\mathrm{emf}k_1 \Dashv L \arrow[r, Rightarrow, "P"] & M \models \mathrm{emf}k_1 \wedge \mathrm{emf}k_2 \arrow[u, Rightarrow, "\text{if } |V_M| \leq k"']
\end{tikzcd}$$

Figure 5.4: Relation on emf and emf$k$

**Fact 20.** For typed graphs $L$ of node size $\leq k$, $L \models \mathrm{emf}k$ iff $L \models \mathrm{emf}$.

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

**Theorem 9 (EMF-model repair & completion).** Let $\mathrm{emf}_1$ be a conjunction of negative EMF constraints, $\mathrm{emf}_2$ a conjunction of negative or universal and preserving EMF constraints.

1. There is an EMF-model repair for $\mathrm{emf}_1 \wedge \mathrm{emf}_2$.

2. There is an EMF-model completion.

3. The EMF-model repair and completion are stable for all typed graphs $L$.

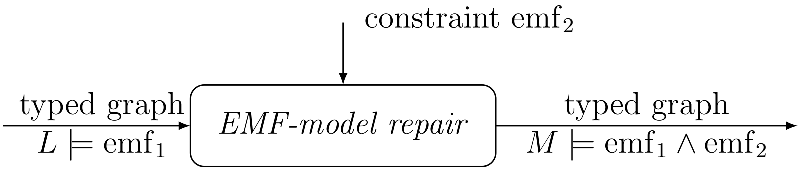

constraint $\mathrm{emf}_2$
typed graph
$L \models \mathrm{emf}_1$
*EMF-model repair*
typed graph
$M \models \mathrm{emf}_1 \wedge \mathrm{emf}_2$

Figure 5.5: Illustration of EMF-model repair

**Remark.** This proceeding also works in practice: In practice, the graphs can have huge sets of nodes. Theorem 9(2) yields an EMF-model completion: For an input graph of node size $k$, construct the EMF$k$ -completion program and apply it to the input graph. This yields an EMF-model completion.

**Proof.** 1. For a typed graph $L$ of node size $k$ satisfying $\mathrm{emf}_1$, we take the EMF$k$ -repair program $P$ for $\langle \mathrm{emf}k_1, \mathrm{emf}k_2 \rangle$ and apply it to $L$. By Fact 20, $L \models \mathrm{emf}_1$ implies $L \models \mathrm{emf}k_1$. By Theorem 3, the application of $P$ to $L$ yields a typed graph $M$ satisfying $\mathrm{emf}k_1 \wedge \mathrm{emf}k_2$. The program does not change the number of nodes, i.e., $M$ is a typed graph with $k$ nodes. By Fact 20, $M$ satisfies $\mathrm{emf}_1 \wedge \mathrm{emf}_2$.

2. For a typed graph $L$ of node size $k$, we take the EMF$k$ -completion program and apply it to $L$ yielding an EMF$k$ -model graph $M$, i.e, a typed graph $M$ satisfying all EMF$k$ constraints. The program does not change the number of nodes, i.e., $M$ is a typed graph with $k$ nodes. By Fact 20, $M$ satisfies $\mathrm{emf}_1 \wedge \mathrm{emf}_2$.

3. By Theorem 8, the EMF$k$ -model repair and completion programs are stable, i.e., for all transformations $L \Rightarrow_P M$, $L \models e$ implies $L \cong M$. Applying the programs to a typed graph, the property remains preserved. □

**Remark (Repair of other structures).** The results in Chapter 4 are applied to meta-modeling: typed graphs are repair w.r.t. EMF constraints. Obviously, typed graphs can also repaired w.r.t. other constraints, e.g., OCL-graph constraints as considered in [RAB$^+$18]. The presented results hold in every $\mathcal{M}$-adhesive category with $\mathcal{E}'$-$\mathcal{M}$ pair factorization. As a consequence, we can do repair for high-level structures and high-level constraints [EGH$^+$14].

**Remark (Model generation).** In model generation, given a meta-model, one tries to find some (all) instances of the meta-model. Model generation may be seen as a special case of model completion applying the program: For a fixed $k$, the application of the EMF$k$ -model completion program to the empty typed graph yields EMF$k$ -model graphs. By Fact 19, every EMF$k$ -model graph with node size $\leq k$ is an EMF-model graph. In this way, we obtain some instances of the meta-model.

## 5.4 Related work

We present some related concepts on model repair, for which there is a wide variety of different approaches. The most significant paper on different model repair techniques and a feature-based classification of these approaches is given by **Macedo et al.** [MTC17].

We start with model repair approaches, which are based on the Eclipse Modeling Framework. The most significant approach is the one by Nassar et al. [NRA17, NKR17]. Another framework is the so-called Diagrammic Predicate Framework (DPF), which is based on the thesis by Rutle [Rut10]. Based on this framework, the model completion approach by Rabbi et al. [RLYK15] and the model repair approach by Wang [Wan16] have been developed, which we present thereafter. Afterwards, we compare some approaches based on the Unified Modeling Language (UML). Finally, we classify our approach according to the features by Macedo et al. [MTC17].

## Eclipse Modeling Framework

In **Nassar et al.** [NKR17, Nas20], a rule-based approach to support a modeler in automatically trimming and completing EMF models and thereby resolving their cardinality violations is proposed. Repair rules are automatically generated from multiplicity constraints imposed by a given meta-model. The rule schemes are carefully designed to preserve the EMF model constraints.

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

(1) Model trimming eliminates supernumerous model elements.

(2) Model completion adds required model elements.

It is shown that both of the algorithms are correct, and, for fully finitely instantiable type graphs, the model completion algorithm terminates.

One can use the approach in this paper to transform a typed graph to an EMF model graph, then, one can use the approach of [NKR17], to transform an EMF model graph to an EMF model graph, satisfying additional multiplicity constraints (see Figure 5.6).

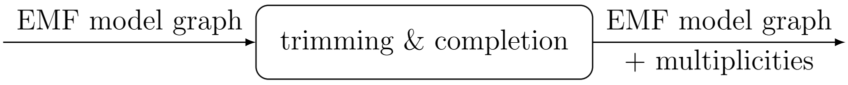


Figure 5.6: EMF model repair in [NKR17]

We compare the approaches in more detail. Both approaches have two phases for the repair steps: they first **delete** all violating constraints, afterwards, the graph is **completed**. In Nassar et al. [NKR17], it is shown that both the model trimming and completion algorithms are **correct**, and the completion algorithm is terminating, provided that for each graph $G$ satisfying the lower bound multiplicity constraints, there exists a graph $G'$, such that $G$ is a subgraph of $G'$, and the type graph meets the “critical edge condition”. Both approaches are **stable** [NKR17, Corollary 2] and Lemma 23. Furthermore, in EMF it is a desired property for models to be *rooted*, i.e., that (at least all non-abstract) nodes are transitively contained in one root node. At the instance level, this enables the creation of a tree structure that can be persisted conveniently. The approach of Nassar et al. is designed to preserve the **rootedness** ability, i.e., under the assumption that the input EMF model graph is rooted, the algorithm yields a rooted EMF model graph again. In our approach, rootedness has not been considered. The reason for this is, that rootedness is not a first-order property. One difference is the kind of constraints, which are considered. The approach of Nassar et al. was designed to preserve the EMF constraints and then repair the multiplicity constraints. In our approach, we used the repair programs to repair the EMF constraints. Furthermore, we have not considered multiplicity constraints explicitly. Our approach can be applied to repair multiplicity constraints, provided that they meet the requirement for Theorem 3. It is up to future work to see if this is feasible for practical applications. It would be interesting to know whether the repair rules and algorithms by Nassar et al. are special cases of our approach in some form.

In **Biermann et al.** [BET12], for EMF-model transformations, consistent transformations are defined. Given a set of rules, the rules are slightly modified, to so-called consistent transformation rules. This way, a direct transformation step applied at an EMF-model graph yields an EMF-model graph again. In our approach, a direct transformation step leads to typed graphs. We use the repair program to complete the typed graph to an EMF model graph.

In **Köhler et al.** [KLT07], graphs with containment edges, and homomorphisms between them are introduced (see end of Section 2.1). The containment condition ensures that a rule can only be applied to an EMF-model graph if the result does not violate any containment constraints. If the first-order containment constraints would be integrated into the rules (as in our approach), a rule with application conditions would be applicable to a typed graph if and only if the typed graph does not violate any first-order containment constraints. We use repair programs (based on rules with application conditions) to repair typed graphs into EMF-model graphs. The main difference is the handling of the non-first-order containment constraint.

In **Taentzer et al.** [TOLR17], a designer can specify a set of so-called change-preserving rules and a set of edit rules. Each edit rule, which yields an inconsistency, is then repaired by a set of repair rules. The construction of the repair rules is based on the complement construction. It is shown that a consistent graph is obtained by the repair program, provided that each repair step is sequentially independent of each following edit step, and each edit step can be repaired. The repaired models are not necessarily as close as possible to the original model.

In **Barriga et al.** [BRH19], an algorithm for model repair based on EMF is presented, which relies on reinforcement learning. For each error in the model, a so-called Q-table is constructed, storing a weight for each error, and repair action. This weight indicates how good a repair action is, depending on the repair action and regarding the user's preferences. The approach can repair errors provided by the EMF diagnostician. The results are evaluated using mutation testing.

In **Kosiol et al.** [KSTZ20] two notions of consistency as a graduated property are introduced: consistency-sustaining rules do not change the number of violations of a constraint in a graph, and consistency-improving rules reduce the number of violations in a graph. The definition is based on the so-called consistency index, given by the number of constraint violations in a graph, divided by the number of "relevant occurrences" of the constraints in a graph. A transformation is consistency sustaining, if the consistency index for the input graph is equal to or less than the resulting graph, and consistency improving if the number of the violations in the resulting graph is smaller than in the input graph. A rule is consistency sustaining if all transformations are. A rule is consistency improving, if all applications of the rule are consistency sustaining, there exists a graph with constraint violations, and a transformation, such that the number of the violations in the resulting graph is smaller than in the input graph. A rule is strongly consistency improving if all its applications to a graph with constraints violating is consistency improving. In their setting, the rules derived from our approach are strongly consistency improving.

In **Hubatschek** [Hub21], a meta-model is formalized as a loop-free type graph with so-called multiplicity constraints, which demand that the number of outgoing edges of the same type for nodes in a model is within certain bounds. It is investigated how the approach to graph repair in this thesis can be used for the repair of multiplicity constraints. It is shown that a repair program for multiplicity constraints of a meta-model exists, if and only if the meta-model has no cycles whose edges all have multiplicity lower bounds of zero.

## Diagrammic Predicate Framework

In **Rutle** [Rut10], the Diagrammic Predicate Framework (DPF) is presented. Models are represented by diagrammatic specifications. A diagrammatic specification consists of an underlying typed graph together with a set of atomic constraints. The typed graph represents the structure of the model and predicates from a predefined diagrammic predicate signature are used to add constraints to the structure. Each predicate has a name, shape, visualisation and semantic interpretation. Additionally, DPF models are built on a stack of typed graphs, where each layer is typed by the previous one.

In **Rabbi et al.** [RLYK15], a model completion approach for predicates specified in the Diagrammic Predicate Framework is introduced. For every predicate in the model, they derive a set of completion rules, by constructing the pullback of the instance, the meta-model and the graph of the condition. These rules, applied as long as possible, yield a model which conforms to the predicate. In our approach, the rules are derived from the constraint and the meta-model. In both approaches, the meta-model remains unchanged.

In **Wang** [Wan16], the semantics of the predicates in the DPF is specified as graph constraints, and a model repair approach for these graph constraints is introduced. For constraints of the form $\forall(L, \exists\, R)$ or $\forall(L, \nexists\, R)$, repair rules are directly derived from the constraints. The construction is based on the construction of subgraphs of $L$ and $R$. For the constraint $\forall(L, \exists\, R)$, for each subgraph $B$, they derive rules $\langle B \Rightarrow R, \nexists\, R\rangle$ and $\langle L \Rightarrow B, \nexists\, R\rangle$. For the constraint $\forall(L, \nexists\, R)$, rules of the form $\langle L \Rightarrow B\rangle$ are derived. The performance of the approach has been optimized for practical application scenarios. In this work, we have combined the programs for proper conditions to a repair program for conjunctions of proper conditions. The properties of the repaired conditions remain preserved, whenever possible. If this is not possible, we delete the occurrence. As far as we can see, the approach in [Wan16] does not handle conjunctive constraints.

In **Sager** [Sag19], a meta-modeling framework based on the DPF with formal constraint semantics and the ability to repair models by existing graph repair approaches, is introduced. They provide constructions for the transformation of models into an equivalent model, where each constraint is on the lowest level (the instance level). They have shown that it is undecidable in general if a program is constraint-preserving.

## Unified Modeling Language

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

In **Puissant et al.** [PSM15], a regression planner is used to automatically generate sequences of repair actions that transform a model with inconsistencies into a valid model. The initial state of the planner is the invalid model, represented as logical formula, the accepting state is a condition specifying the absence of inconsistencies. Then, a recursive best-first search is used to find the best suitable plan for resolving the inconsistencies. The correctness of the algorithm is not proven, but the approach is evaluated through tests on different UML models. The algorithm takes the decision of the user into account and uses backtracking, to find "the best" suitable plan to repair the inconsistencies. In contrast, our approach is purely automatic.

In **Meier and Winter** [MW18], an approach to synchronize models by integrating models with their meta-models into one integrated (meta)-model is presented. Both approaches formalize consistency rules. Special operators are introduced, which are used to keep all models consistent to each other. These are special operators that take a meta-model and a model as input and returns another meta-model and model. These operators form a chain of operations, which subsequently yield the desired meta-model. In our approach, these operators may be formalized as double-pushout rules. In Meyer and Winter, the consistency rule may be given by a stakeholder.

In **Meyer et al.** [MKW20], an approach to define new viewpoints and views by using operators is introduced. These operators split the whole transformation between one integrated (meta)-model and view(point) into parts. They are designed to be generic and reusable by using meta-model and model decisions. The exemplary chains which are presented show their reusability for different viewpoints on top of the already existing integrated meta-model.

## Feature-based classification

In Macedo et al. 2017, the model repair approaches are classified into the main features (1) *domain*, the (2) *constraint*, the (3) *update* of a user, the (4) *check*, and the (5) *repair*. In the following, we classify the mandatory features from Macedo et al. for our approach.

(1) **Domain.** In our approach, the domain, i.e., the model domain space, is formalized as *graph*. The *technical space* of our graph repair approach from Chapter 4 is independent of any domain. In this chapter, we have applied the theory to meta-models specified in the Eclipse Modeling Framework. Following the graph-based approach, our repair programs are *meta-model independent*, i.e., we can represent any meta-model as a typed graph, and repair it to a typed graph satisfying the constraints.

(2) **Constraint.** Model repair approaches may optionally expect a constraint to be accompanied with repair hints on how to generate repair updates. Techniques where the repair procedures automatically derive the repair updates from the constraint (for example rule-based approaches) do not have this kind of repair hints. The *shape* of our constraints may be seen as pattern matching, equipped with negative application conditions.

(3) **Update.** The user updates may either be *state-based* where the repair procedure only considers the post-state of the user update, or *delta-based*, which require information regarding the user actions that led to the current state of the environment. Our approach considers only the condition $d$, which may be seen as the post-state of the user-update. Thus, our approach may be seen as a state-based approach.

(4) **Check.** In our approach, we do not have a check-only mode, i.e., we use a *coupled* grammar-based approach, that is, we use repair rules and the checking procedure of the violations is coupled to the repair rules as a precondition. These

rules cannot be applied in *check-only* mode.

(5) **Repair.** The core of our approach is *rule-based*, i.e., we apply a set of rules, whenever a condition is violated. *Syntactic* techniques automatically derive repair plans by syntactic analysis of the constraints. These repair plans are calculated at static time and then instantiated to concrete model instances at run-time. This applies to our approach: we derive a repair program syntactically from the constraint, afterwards, we apply the program to a typed graph representing the instance. *Incremental* approaches reuse data from previous checking of repair procedures. This does not apply to our approach. For repair procedures, one important research question is the kind of semantic properties, which are guaranteed by the repair procedure. A technique is *total* if, for every user update that results in an inconsistent state, it is able to produce a repair update. The solid (destructive) repair programs for legit (satisfiable) conditions are applicable to any input graph, and thus, may be seen as total. A repair procedure is *fully consistent* if it guarantees that the number of inconsistencies is at a minimum. We have proven that the repair programs are *correct* (Theorem 5), that is, after the application of the program the graph satisfies the constraint, i.e., it is fully consistent. A repair procedure is stable if it does not change the input if the constraint is already satisfied. The solid repair programs are stable (Lemma 23). A repair procedure is *least-changing* if the repaired models are as close as possible to the original. The solid repair programs are maximally preserving (Lemma 24).

## 5.5 Conclusion

In this chapter, we have applied the theory of typed repair programs to EMF$k$ - and EMF-model repair. The aim is to apply the theory, developed for first-order graph formulas, to get EMF-model repair. We have considered EMF$k$ constraints, a first-order variant of EMF constraints, as a helper construction. Finally, we have used the results on EMF$k$ -model repair to get EMF-model repair. An illustration is given in Figure 5.7.

Summarizing, we have

(1) **EMF$k$ -model repair.** For EMF$k$ constraints, a first-order variant of EMF constraints, we present stable EMF$k$ -model repair and completion programs. Application of these programs to any typed graph yields a repaired typed graph and an EMF$k$ -model graph, respectively,
(2) **EMF-model repair.** These results are applied to the EMF world and yield to EMF-model repair and completion results.

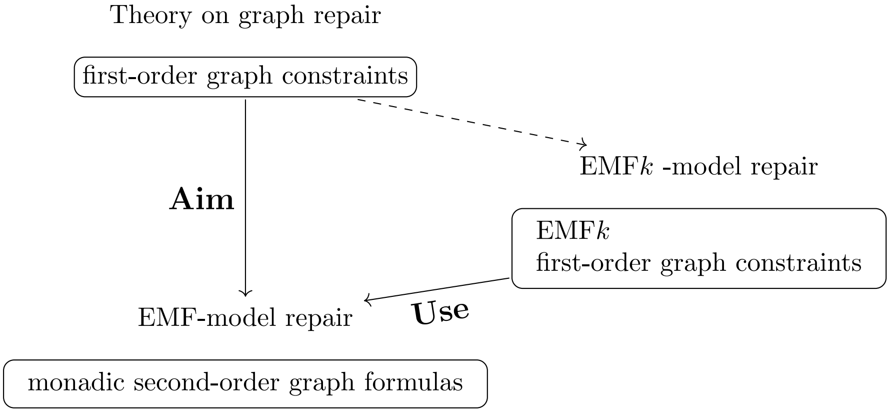

Theory on graph repair
first-order graph constraints
Aim
EMFk -model repair
EMFk
first-order graph constraints
Use
EMF-model repair
monadic second-order graph formulas

Figure 5.7: Application of the theory on graph to EMF-model repair

## Further work

It remains the question, how far this approach is applicable in practice. Interesting aspects may include the kind of constraints that are needed (e.g. multiplicity constraint as in Nassar et al. [Nas20]), the complexity of the algorithms.

# Chapter 6

# An implementation in ENFORCE$^+$

ENFORCE is a system to ensure formal correctness of graph programs (Azab et al. [AHPZ06]). It implements many of the needed tools, such as graph conditions, graph programs, and those to verify correctness of graph programs. In this chapter, we extend ENFORCE by a component for graph repair, calling the result ENFORCE$^+$ (see Figure 6.1), give an overview of the implemented repair component and show some implementation details.

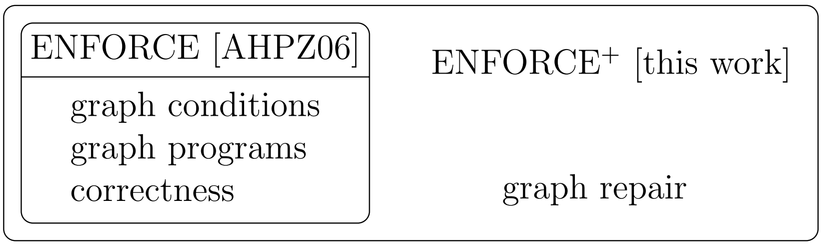


Figure 6.1: ENFORCE and ENFORCE$^+$

The idea of the system ENFORCE$^+$ is to automatically construct a repair program and check whether the repair program constructed per hand is correct. For this purpose, we restrict to directed, labelled graphs. It would be nice to have full implementation of typed graph repair (see Further Topics).

The structure of the chapter is as follows. In Section 6.1, we describe the tool ENFORCE$^+$, while in Section 6.3 we describe the implementation of graph repair in ENFORCE$^+$, give some details on its structure and implementation and show an example of its usage. An overview of related concepts and implementations is given in Section 6.4. In Section 6.5, we draw some conclusions.

## 6.1 ENFORCE$^+$

ENFORCE is a framework for verifying graph program specifications. It was originally written in Java 6 and provides suitable data structures, for instance, graphs, categories, morphisms, conditions, rules, programs, as well as operations on these structures, such as enumeration of all epimorphisms for a given domain, a construction of weakest liberal preconditions, and so forth. The basic design of ENFORCE is presented in Azab et al. [AHPZ06].

The system ENFORCE$^+$, an extension of ENFORCE, is a framework for graph repair. In the system, there are two main components:

1. Construction of repair programs
2. Application of a repair program to a graph (called "repair").

To implement graph repair in ENFORCE$^+$, we enhanced ENFORCE with some additional features. Table 6.1 gives an overview of features of ENFORCE and features in ENFORCE$^+$.

Table 6.1: Features of ENFORCE and additional features for graph repair

| Features in ENFORCE | Extensions and additions |
|---|---|
| Graph conditions | |
| Shift-construction | |
| Programs with left partial interface | Programs with interfaces |
| Application of programs with left partial interface | Application of programs with interfaces |
| | `try` statement |
| | Check for proper conditions |

## 6.2 Design decisions

The construction of a repair program takes a graph condition as input and creates a repair program for the condition if possible. Otherwise, if the shape of the condition is not one of those specified in Theorems 2 and 3, an exception is thrown since it is undecidable if a sequence of conditions is preserving [Sag19]. The input is given in a textual form while the output may be graphical (or textual).

**Input.** The input is a graph condition specified in Java.
**Output.** The output (a repair program or repaired graph) may be rendered to PDF via LaTeX (see Example 57), or given in textual form.

The following simplifications are made in the output:

1. Rules $\varrho = \langle x, p, \mathrm{ac}, y\rangle$ (*with interfaces* $X, Y$). If both interfaces $X, Y$ are empty, we write $\varrho = \langle p, \mathrm{ac}\rangle$. If additionally $\mathrm{ac} = \texttt{true}$, we write $\varrho = \langle p\rangle$ or short $p$.

2. Programs. Every rule $\varrho$ with interface $X$ (and $Y$) is in $\mathrm{Prog}(X)$ (no brackets)

The repair programs are composed of rules with interface morphisms. Internally, repair programs are represented with their interface morphisms. As output, one may get the full version with interface morphisms, as well as a more readable version without interfaces (instead of the interface morphisms). For our purposes, the long form of graph programs can be inferred from the short form since the left interface usually coincides with the left graph of rules and the right interface usually coincides with the left interface (to enable iteration). Thus, no information is lost.

**Notation.** Given a rule $\varrho = \langle x, p, \mathrm{ac}, y\rangle$ with plain rule $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$, the short form of $\varrho$ is obtained as follows:

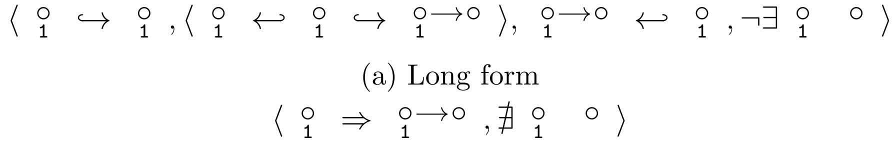
1. Abbreviate $p$, i.e., $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$ becomes $p = \langle L \Rightarrow R\rangle$.

2. Abbreviate negations, i.e., write $\neg Q$ in ac as $\not{Q}$, where $Q \in \{\forall, \exists\}$.

3. If $\mathrm{ac} = \texttt{true}$, remove ac.

**Remark (interfaces not drawn).** The interfaces of a rule are represented and used by the system. In the short version of a rule $\langle x, p, \mathrm{ac}, y\rangle$, the interfaces $x$ and $y$ are not shown. For better understanding of the graphical output, it would be nice, to represent the interfaces $x\colon X \hookrightarrow L$ and $y\colon Y \hookrightarrow R$ in the output, e.g., by marking or coloring the nodes of $X$ and $Y$ in the graphs $L$ and $R$, respectively.

**Example 57 (graphical output).** In the following, we show the long and short form of the graphical output of ENFORCE$^{+}$. The long form of the rule, shown in Figure 6.2a, shows both interfaces of the rule and uses $\neg\exists$ in the application condition of the rule. In the short form of the output shown in Figure 6.2b, the left and right interfaces are omitted (but can be inferred) and $\neg\exists$ is abbreviated to $\nexists$.



(a) Long form

(b) Short form

Figure 6.2: ENFORCE$^+$'s graphical output modes

The following design decisions have been made with regard to the implementation.

**Decision (Directed, labelled graphs).** By [EEPT06a, Fact 2.9], labelled graphs can be seen as special typed graphs. The implementation considers repair programs for directed, labelled graphs instead of typed graphs. The reason for this is that ENFORCE is implemented for directed, labelled graphs.

**Decision (Properness check).** There is an algorithm to check properness of a condition which returns true if the condition is proper, and false otherwise. The check is used before constructing a repair program. If the input condition is proper, a repair program is created according to Construction 13. Otherwise, an `IllegalArgumentException` is thrown, informing the user why the condition is not proper, and no repair program is created.

**Decision (Preservation check).** Checking whether a conjunctive condition is preserving is in general undecidable (see Theorem 6, [Sag19]); decidable cases (apart from all subconditions negative (positive)) have not yet been identified. The question is deferred to the user (but has to be answered a priori), they can specify whether preservation is known or unknown. In the first case, the system constructs a program and creates a message “Assuming preservation (requested by user)”, in the second case a warning “Preservation status unknown, program may not be a repair program” is emitted.

**Decision (Dangling edges).** The dangling-edges operator instructs to apply a rule as in the SPO-approach, i.e., if a node is to be deleted, to delete dangling edges attached to the node and then remove the node itself. Since ENFORCE supports only the DPO-approach, we simulate the SPO-approach via a program that marks the nodes to be deleted, removes dangling edges, removes the nodes and finally performs the unmarking.

**Decision (Invariant approximation).** ENFORCE$^+$ implements as long as possible iteration of a program $P$ as $\langle P^*; \texttt{Mark}(\mathrm{id}_C, \mathrm{Wlp}(P, \texttt{false}))\rangle$, where

Wlp$(P, c)$ for a program $P$ and condition $c$ denotes the weakest liberal precondition [Pen09]; this guarantees the existence of results (see Fact 2, [HPR06]). Since in general, the weakest liberal precondition is unbounded, the computation may not terminate. We use invariant overapproximation to compute an approximation of a finite representation of the weakest liberal precondition (cf. [Pen09]) during the execution of programs to obtain termination of the computation.

## 6.3 Graph repair in ENFORCE$^+$

In this section, we outline how graph repair has been implemented in ENFORCE$^+$ and provide some examples of the application of ENFORCE$^+$. The section is divided into three parts: first, the handling of proper conditions (without conjunctions and disjunctions) is described and second the handling of conjunctions in graph repair is described. Finally, we summarise the features of our implementation.

### Graph repair for proper conditions

Graph repair is supported in ENFORCE$^+$ as a repair component in its `transformations` package (see Figure 6.3).

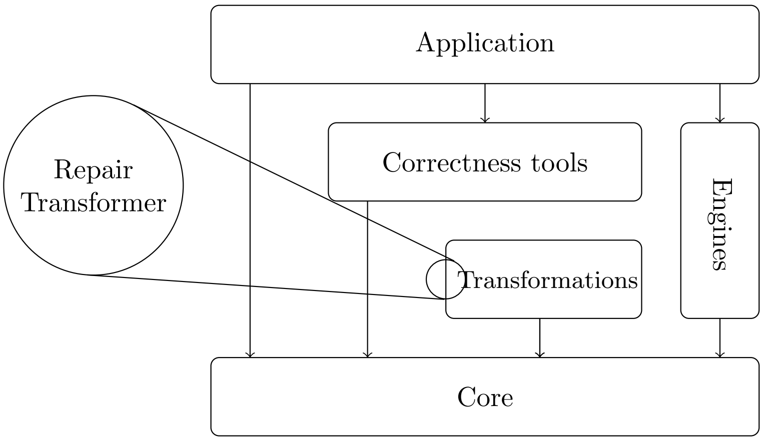


Figure 6.3: Integration of graph repair in ENFORCE$^+$

Figure 6.4 shows the structure of our graph repair implementation in a UML class diagram. The main part of the repair component is the class `ProperRepairTransformer`, which contains a method `transform(Condition d)` returning a repair program for the condition according to Construction 13, or throwing an `IllegalArgumentException` if the condition is not proper. The implementation of Theorem 2 in `ProperRepairTransformer` is straightforward due to ENFORCE$^+$'s simple and highly readable graph program implementation, and the inductive definition of Theorem 2.

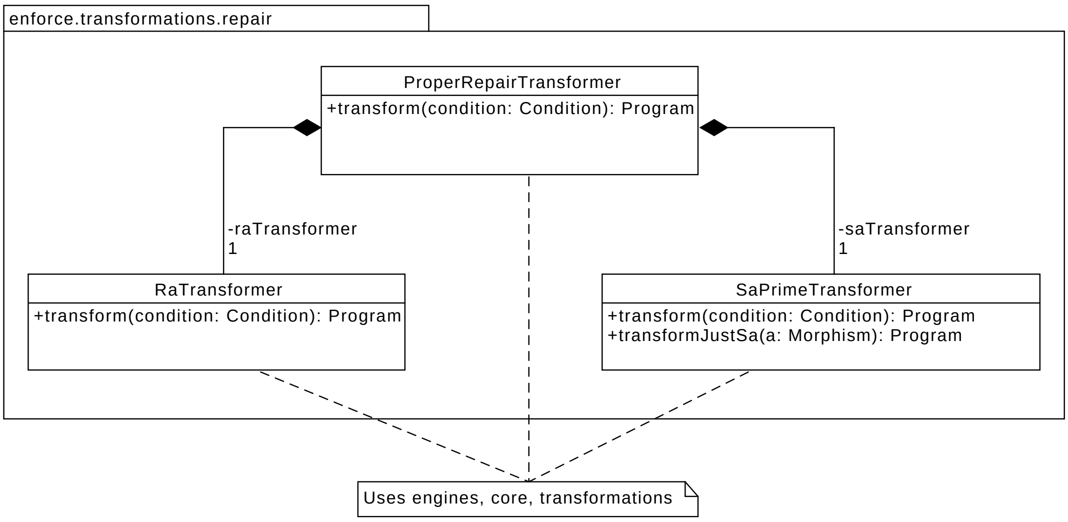

enforce.transformations.repair
ProperRepairTransformer
+transform(condition: Condition): Program
-raTransformer
1
-saTransformer
1
RaTransformer
+transform(condition: Condition): Program
SaPrimeTransformer
+transform(condition: Condition): Program
+transformJustSa(a: Morphism): Program
Uses engines, core, transformations

Figure 6.4: Class diagram of our repair component for proper conditions in ENFORCE$^+$

To construct a repair program, the construction of repairing sets $\mathcal{R}_a$ and $\mathcal{S}_a$ for morphisms $a\colon A \hookrightarrow C$ is essential.

Repairing sets $\mathcal{R}_a$ and $\mathcal{S}_a$ as well as the program $\mathcal{S}'_a$ are implemented in `RaTransformer` and `SaPrimeTransformer`, respectively. `RaTransformer` has a method `transform(Condition d)` which returns $\mathcal{R}_a$, `SaPrimeTransformer` has a method `transform(Condition d)` which returns $\mathcal{S}'_a$, and a method `transformJustSa(Morphism a)`, which returns $\mathcal{S}_a$. The latter rule set plays a part in the implementation of Theorem 3.3.

Figure 6.5 shows the implementation of the `RaTransformer.transform` method in pseudocode form. The loop in line 4 iterates over all subgraphs between $A$ and $C$. Line 5 ensures that the repairing set does not contain identity rules. Line 7 creates the application condition of the rule under creation, while lines 8 and 9 define the rule's left, right morphisms and left, right interface morphisms, respectively. Lines 10 and 11 create the rule instance from the morphisms and add it to the repairing set. The repairing set is returned in line 13.

For the creation of application conditions (line 15), a conjunction is created. The shifted $\nexists a$ condition is computed in line 17, while lines 18 through 22 implement the termination condition $\mathrm{ac}_B$, i.e., the conjunction of $\nexists(B \hookrightarrow B')$ for all subgraphs $B'$ between $B$ and $C$, unless $B'$ is isomorphic to $B$. The resulting conjunction is returned at the end of the procedure.

```
function transform(Condition d)
    ruleSet = {}
    A := dom(d.morphism), C := codom(d.morphism)
    foreach B in graphsBetween(A, C) do
        if $B \cong C$ then continue

        applCond := createApplCond(d.morphism, A, B, C)
        right := B $\hookrightarrow$ C, left := B $\hookrightarrow$ B
        leftInterface := A $\hookrightarrow$ B, rightInterface := d.morphism
        rule = Rule(leftInterface, left, right, rightInterface, applCond)
        ruleSet.add(rule)
    end
    return ruleSet

function createApplCond(Morphism a, Graph A, B, C)
    condition = new Conjunction
    condition.add(Shift(A $\hookrightarrow$ B, $\nexists a$))
    foreach B' in graphsBetween(B, C) do
        if $B' \cong B$ then continue

        condition.add($\nexists(B \hookrightarrow B')$)
    end
    return condition
```

Figure 6.5: Pseudocode for repairing set $\mathcal{R}_a$

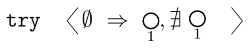
**Example 58 (application of ENFORCE$^+$).** In the following examples, we use ENFORCE$^+$ to generate repair programs for several conditions. The code snippet in Java below is used to generate a PDF representation of the repair programs. Line 1 defines the condition $d$ in each case, line 2 instantiates a `ProperRepairTransformer`, line 3 constructs the repair program for $d$, and finally line 4 renders the repair program to PDF via LaTeX.

```
1    Condition d = ... //construct input condition;
2    ProperRepairTransformer repairTransformer = new
        ProperRepairTransformer();
3    Program program = repairTransformer.transform(d);
4    TeXScreenRenderer.renderTeX(program, "example");
```

1. For the constraint $d = \texttt{true}$, the repair program is

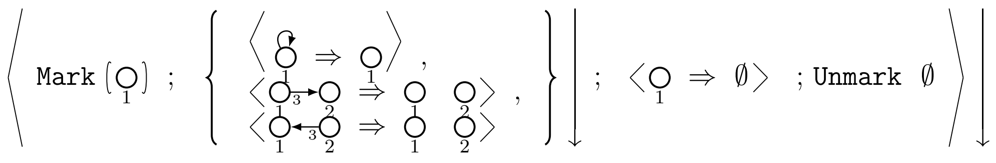
   `Skip`

2. For the constraint $d = \exists\ \circ$ , the repair program is

   The program works by creating a node, provided that there does not exists one.

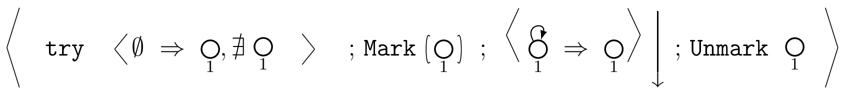
3. For the constraint $d = \nexists\ \circ$ , the repair program is[1]

   The program deletes a node by simulating the SPO-approach, i.e., it marks a node, removes dangling edges from the node as long as possible, removes the node and finally unmarks it.

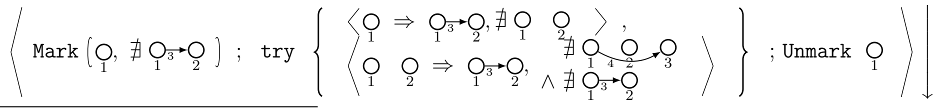
   **Remark.** Instead of presenting the program for deleting dangling edges, one can use the dangling edges operator to indicate it.

4. For the constraint $d = \exists(\ \circ\ , \nexists\ \circ\!\circlearrowright\ )$, the repair program is

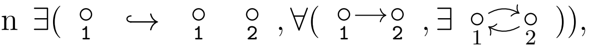
   

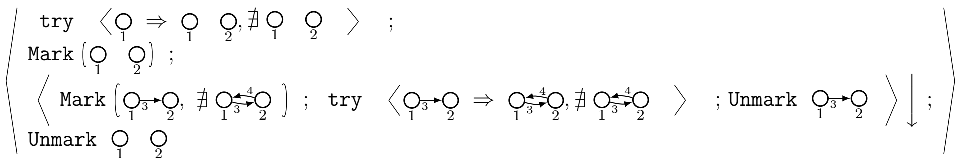
   The program creates a node, provided that there does not exist one, marks a node, as long as possible removes loops from the marked node, and finally unmarks the node.

5. For the constraint $d = \forall(\ \underset{1}{\circ}\ , \exists\ \underset{1}{\circ}\!\longrightarrow\!\circ\ )$, the repair program is

[1] The program for deleting dangling edges seems to have isomorphic rules. However, this is due to the short version of the rule, where the interfaces are not shown (see Remark (interfaces not drawn) in Section 6.2).

6. For the condition , the repair program is



The program works by adding another node, provided that there is none and then marks two nodes. As long as possible, two nodes and an edge between them are marked, provided that there is no opposite edge, then the opposite edge is added, provided that it does not exist yet, and the previously marked nodes and the edge are unmarked. Finally, the two nodes marked in the beginning are unmarked.

This program cannot be applied to a graph where the domain of the input morphism is empty, because it is a program with a node as the interface. The program contains application conditions, which are checked twice in the `Mark` statement, and in the application condition of the rule. This is a very special case, due to the simplicity of the condition, and can, in general, not be omitted.

**Remark.** In very special cases, conditions are checked twice: for the condition $d = \forall(o, \exists\, a)$, the repair program is $P_d = \langle \texttt{Mark}(o, \nexists\, a); \texttt{try } \mathcal{R}_a; \texttt{Unmark}(o)\rangle\downarrow$. The first rule in $\mathcal{R}_a$ is the rule with left-hand side $A$ and application condition containing $\mathrm{Shift}(\mathrm{id}_A, \nexists\, a) \equiv \nexists\, a$. Thus, the condition $\nexists\, a$ is checked by $\texttt{Mark}(o, \nexists\, a)$ as well as in the application of the first rule in $\mathcal{R}_a$. (Construction 13 can be modified in the way that, for $d = \forall(o, \exists\, a)$, the subprogram $P_{\exists a} = \texttt{try } \mathcal{R}_a$ is replaced by the program $P^-_{\exists a} = \texttt{try } \mathcal{R}^-_a$ where $\mathcal{R}^-_a$ is obtained from $\mathcal{R}_a$ by replacing the first rule by the rule without the application condition $\nexists\, a$.) For a condition $d = \forall(o, c)$ with $c = \exists\,(a, c')$, the repair program is similar: $P_d = \langle \texttt{Mark}(o, \neg c); \texttt{try } \mathcal{R}_a; \ldots; \texttt{Unmark}(o)\rangle\downarrow$. Then the condition $\neg c = \nexists\,(a, c')$ as well as the application condition $\nexists\, a$ of the first rule in $\mathcal{R}_a$ is checked. Since, generally, $\nexists\,(a, c') \not\Longrightarrow \nexists\, a$, both conditions have to be checked. We do not consider these simplifications further since they provide only insignificant runtime improvements.

## Conjunctive repair

Repair of conjunctive constraints according to Theorem 3 is implemented in a class `ConjunctionRepairer` (see Figure 6.6) which has a function

`transform(Conjunction conjunction, PreservationMode mode)`, where the second parameter is optional, defaulting to `UNKNOWN`. It makes use of the component

ProperRepairTransformer to generate the repair programs needed to construct repair programs for supported conjunctions.

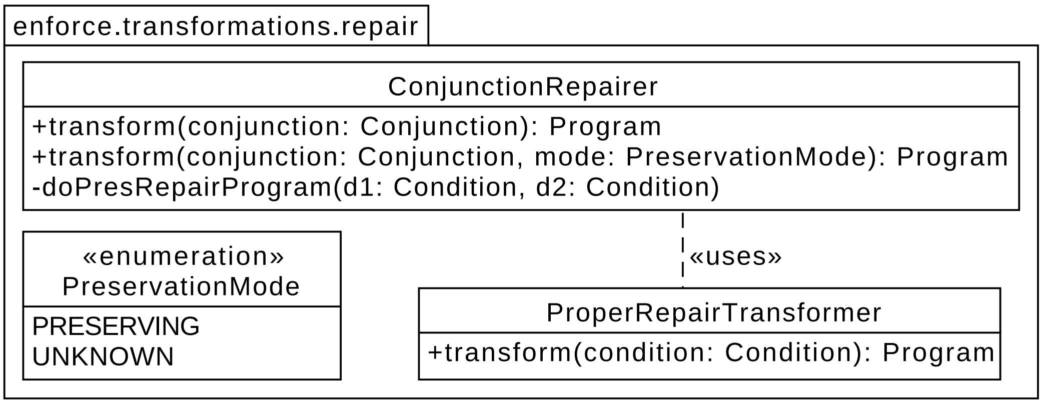

enforce.transformations.repair
ConjunctionRepairer
+transform(conjunction: Conjunction): Program
+transform(conjunction: Conjunction, mode: PreservationMode): Program
-doPresRepairProgram(d1: Condition, d2: Condition)
«enumeration»
PreservationMode
PRESERVING
UNKNOWN
«uses»
ProperRepairTransformer
+transform(condition: Condition): Program

Figure 6.6: Class diagram of the conjunction repairer

Algorithm 6.7 shows the pseudocode for the implementation of repair programs for conjunctive conditions. Line 2 first flattens[2] the conjunction if necessary, and then decomposes the conjunction into lists of positive, negative and universal conditions, respectively. Lines 3 through 18 make use of the different condition shapes which are available in the previously decomposed conjunction, and calls the relevant subroutines to create repair programs in each case. If no specific pattern could be discerned in the conjunction, then the preservation mode provided by the user is taken into account, a warning is printed to the console, and the condition is handled as if it consisted of a preserving sequence of subconditions, i.e., the repair programs for each subcondition are sequentially composed. Conjunctions of positive or negative conditions are handled similarly, but no warning is printed to the user because the subconditions are preserving in those cases. This is different for conjunctions of negative and universal & preserving conditions, where the warning about the preservation mode is printed and the subroutine doNegativeAndUniversal (see Algorithm 6.8) is called, which uses the ProperRepairTransformer to construct the repair programs for negative conditions, and then constructs the negation-preserving repair program for the universal & preserving conditions, according to Construction 16.

[2] Flattening a conjunction pulls any conjunctive subconditions of it up a level.

```
function transform(Conjunction conjunction, PreservationMode mode)
    checkResults := checkShape(conjunction) // flatten,
        sequentialize, separate
    switch checkResults.shape do
        case NEGATIVE do
            return doNegative(checkResults.negativeConditions)
        end
        case POSITIVE do
            return doPositive(checkResults.positiveConditions)
        end
        case NEGATIVE_AND_UNIVERSAL do
            printPreservationMode(mode)
            return
             doNegativeAndUniversal(checkResults.negativeConditions,
              checkResults.universalConditions)
        end
        otherwise do
            printPreservationMode(mode)
            return doPreserving(checkResults.allConditions)
        end
    end
```

Figure 6.7: Pseudocode for conjunctive repair

Algorithm 6.8 line 10 shows the pseudocode for the function `doPresRepairProgram(Condition d1, Condition d2)`. Lines 11 and 12 construct the repair program for `d2` and the `d1`-preserving version of the repair program for `d2`, respectively. The repairing set $\mathcal{S}_a$ and the set $\mathcal{S}_a^{\text{id}}$ are created in lines 13 and 14, respectively, where a function `constructSaId` is used to construct $\mathcal{S}_a^{\text{id}}$. To construct the version of $\mathcal{S}_a^{\text{id}}$ which deletes dangling edges, the `SaPrimeTransformer` is used in line 15. The lines 16 to 18 construct the sequential composition of the selection of elements which violate `d2`'s inner subcondition and $\mathcal{S}_a'^{\text{id}}$. The program $P_{\nexists a}^{\text{id}}$ is created in line 19, by the as-long-as-possible iteration of $\mathcal{S}_a^{\text{id}}$, and in line 20, the result of the whole construction (the sequential composition of `P2d1` and `P_nexists_a_id`) is returned. The preservation check for universal and preserving sequences of conditions is up to the user (see Section 6.2).

```
function doNegativeAndUniversal(Conjunction negatives, Conjunction
 universals)
    seq = Sequence()
    foreach negative in negatives do
        seq.add(ProperRepairTransformer.transform(negative))
    end
    foreach universal in universals do
        seq.add(doPresRepairProgram(negatives, universal))
    end
    return seq

function doPresRepairProgram(Condition d1, Condition d2)
    P2 := ProperRepairTransformer.transform(d2)
    P2d1 := HlrcTransformations.transformPres(P2, d1)
    $S_a$ := SaPrimeTransformer.transformJustSa(d2)
    Sa_id := constructSaId($S_a$)
    Sa_id_prime := SaPrimeTransformer.transformSaPrime(S_a_id)
    seq := Sequence()
    seq.add(Sel(d2.morphism, ¬d2.innerCondition))
    seq.add(Sa_id_prime)
    P_nexists_a_id := AsLong(seq)
    return Sequence(P2d1, P_nexists_a_id)
```

Figure 6.8: Pseudocode for preserving repair program

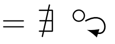
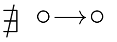
**Remark.** The order of the conditions is implicitly changed during the preprocessing unless the conditions are assumed to be preserving. This is because Theorem 3.3 needs to handle negative conditions before the universal conditions. The order of the conditions within their groups, however, is unchanged.

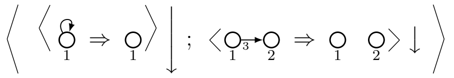
**General Remark.** The following Examples are constructed to illustrate the constructions of the programs. They were designed to be as small as possible, meaning that they contain few rules and that the application conditions only contain very few graphs. The consequence of this is, that conditions may further be optimized (see Example 59.3) and that the programs are simple and may be further optimized: many of the programs contain multiple isomorphic rules and application conditions, which are checked twice, and programs, which may never be applied (see Example 59.4). In a further version, one can optimize the programs such that several conditions do not have to be checked twice. In general, these problems cannot be avoided since, in general, it cannot be decided if two rules are semantically equivalent or if two conditions are semantically equivalent.

**Example 59 (conjunctive repair).** We apply ENFORCE$^+$ to generate repair programs via the following Java code.

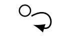
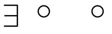
```
1     Conjunction d = //construct input conjunction;
2     PreservationMode mode = PreservationMode.PRESERVING;
3     ConjunctionRepairer conjunctionRepairer = new
         ConjunctionRepairer();
4     Program program = conjunctionRepairer.transform(d,
         mode);
5     TeXScreenRenderer.renderTeX(program, "example");
```

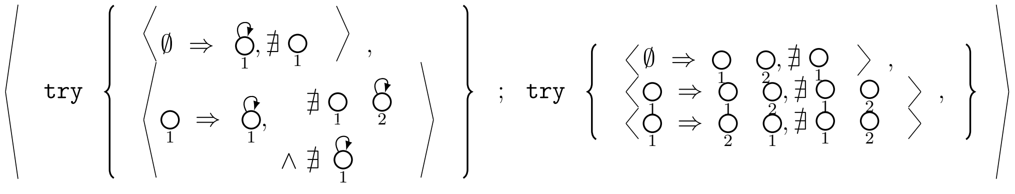
Line 1 creates the condition $d$ and line 2 defines the preservation mode as `PRESERVING`. Line 3 instantiates the `ConjunctionRepairer`, which is used in line 4, to create the repair program for $d$, and in line 4, the PDF output for the generated program is printed.

1. Consider the condition $d = d_1 \wedge d_2$, where $d_1 = \nexists$  and $d_2 = \nexists$  . Then the repair program is

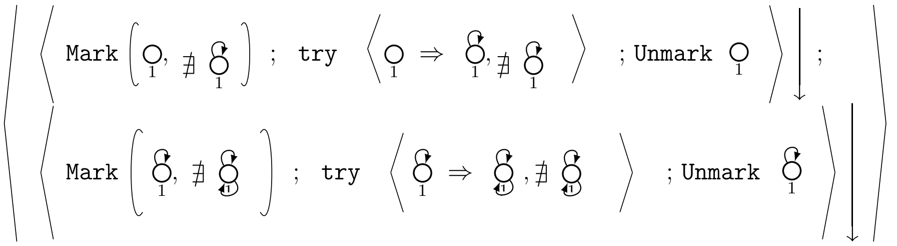
   The program removes a loop from a node as long as possible. Afterwards, any edge between two nodes is deleted as long as possible.

2. Consider the condition $d = d_1 \wedge d_2$, where $d_1 = \exists$  and $d_2 = \exists$  . Then $d_2$ is $d_1$-preserving and the repair program is

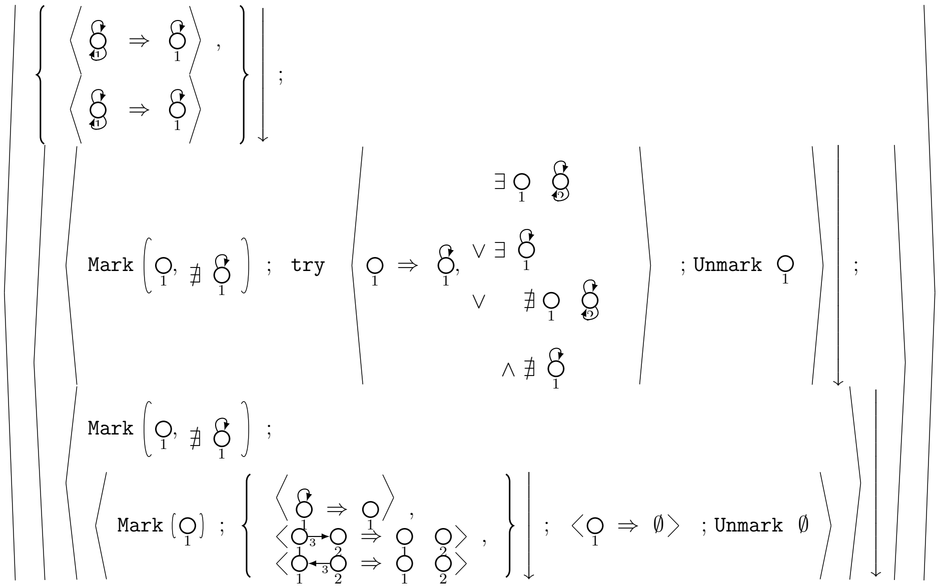
   The program first tries to add a loop by adding a loop or node with a loop, provided there does not exist one. Afterwards, it tries to add two nodes, provided there does not exist one, or adds a node, provided that there exists one but not two nodes. In the repair program for $d_2$, there are two rules containing isomorphic graphs because one rule is created for each of the nodes, which has to be added. The reason is the consideration of all subgraphs in the construction, and the simplicity of the Example. This cannot be avoided since, in general, it cannot be decided if two rules are semantically equivalent.

3. Consider the condition $d = \forall(\ \circ\ , \exists\ \circ\circlearrowright\ ) \wedge \forall(\ \circ\circlearrowright\ , \exists\ \circlearrowleft\circ\circlearrowright\ )$. This condition is equivalent to the linear condition $\forall(\ \circ\ , \exists\ \circlearrowleft\circ\circlearrowright\ )$ and the repair program contains application conditions, which are checked multiple times. We have chosen this condition to illustrate the construction of the repair programs for universal and preserving sequences of conditions. Then the repair program is



The program first creates a loop at every node, provided that the node does not have a loop and then creates another loop at every node with a loop, provided that it does not have two yet.

4. Consider the condition $d = \nexists\ \circlearrowleft\circ\circlearrowright\ \wedge \forall(\ \circ\ , \exists\ \circ\circlearrowright\ )$. Then the repair program is



The program first as long as possible removes a loop from each node that has two loops. Then, as long as possible, it attaches a loop to a node, provided that it does not have a loop. Finally, any node without a loop is removed by simulating the SPO-approach via a subprogram to delete dangling edges. This example is constructed, to illustrate the construction of the repair programs for negative and universal and preserving sequences of conditions. Thus, the program contains two isomorphic rules. The reason is the consideration of all subgraphs in the construction, and the simplicity of the Example. Since the universal condition preserves the negation, the decreasing program may never be applied.

**Remark.** For the implementation, our aim was to construct the correct repair programs. Of course, there are some optimizations for future work. These may include checks for isomorphic rules, and further simplification of the application conditions, e.g., as in Nassar et al. [NKAT20].

### Feature summary

Based on the above examples, the features of our implementation can be summarised as follows. ENFORCE$^+$'s graph repair component can be used to automatically repair proper conditions and some conjunctive conditions according to the theory. Hence, the implementation generates graph programs that are *stable*, *maximally preserving* and *terminating*. User interaction is limited to providing the conditions to construct repair programs for, and the preservation mode for conjunctive constraints. At the moment, it is not possible for the user to pause the repair process and, for instance, select specific matches at runtime.

## 6.4 Related systems

In this section, we give an overview of some closely related systems as well as some general graph transformation tools. A sophisticated survey and a feature-based classification of model repair approaches can be found in [MTC17].

### Henshin/EMF Model Repair

Henshin is a transformation language and tool environment for attributed graph transformation for the Eclipse Modeling Framework (EMF). Among its features are mixing of the double and single pushout approach to graph transformation, a graphical editor, and critical pair analysis. It supports various editors, execution engines and toolchains (see Strüber et al. [SBG+17]). At the moment, ENFORCE+ does not support EMF or attributed typed graphs. In **Nassar et al.** [NRA17, NKR17], a rule-based approach to guide modelers in an automated, interactive way in the setting of model repair in EMF is presented. The authors give a rule-based algorithm that works in two steps, *model trimming* and *model completion*. The algorithm assumes the existence of a rule-based model transformation system, which the authors describe how to derive from a given meta-model. The approach is implemented in two Eclipse plug-ins based on Henshin, where the resolution strategy is semi-interactive, and the tool guides modellers to valid models. In contrast, ENFORCE+ only interacts with the user when preservation of conjunctive constraints cannot be handled directly due to unknown preservation status, where the user is warned and asked to provide the status. Another difference is that the Henshin implementation allows moving actions, i.e., violation of upper bounds of multiplicities may be resolved by moving items to a compatible node without exceeded multiplicity constraint, or such a node may be created first and then the items moved [NKR17]. This is currently not supported in ENFORCE+. Since our tool focuses on graph repair instead of model repair, the implementation does not depend on a given meta-model, but the conditions to repair are supplied by the user directly as graph conditions.

## AutoGraph

In **Schneider et al.** [SLO18], a logic of attributed graph properties, where the graph and attribution part are separated, is introduced. This is accompanied by a dedicated implementation called Autograph, written in Java. In Schneider et al. [SLO19], a logic-based incremental approach to graph repair is presented, generating a sound and upon termination complete overview of least changing repairs. The graph repair algorithm takes a graph and a first-order graph constraint as inputs and returns a set of graph repairs. Given a condition and a graph, they compute a set of symbolic models, which cover the semantics of a graph condition. Both approaches are proven to be correct, i.e., the repair (programs) yield to a graph satisfying the condition. The delta-based repair algorithm in [SLO19] is a dynamic approach since it takes the graph update history into account. This is in contrast to our approach which statically generates a repair program for a given graph condition, upon which ENFORCE+'s repair component is based. The approach implemented in ENFORCE+ is furthermore guaranteed to terminate, whereas the approach in [SLO19] may not terminate if repair updates influence each other.

## VIATRA-Solver

The VIATRA-Solver of **Semaráth et al.** [SNV18], is an open-source tool and graph solver for the automated generation of consistent, domain-specific models, which is available as an Eclipse plugin, a standalone Java application or an API. The basis of the approach are partial models. Formally, an initial partial model is refined via so-called decision- and unit-propagation rules (which are derived in a pre-processing step from a given meta-model), reducing the number of uncertainties and violations, to converge to a valid model. Thus, our approach is different in that labelled, directed graphs are the underlying structure and that the actual repair process is encoded directly via graph programs; no exploration is used to find valid models. In contrast to ENFORCE$^+$, VIATRA-Solver used a meta-model to extract constraints, whereas the constraints must be supplied to ENFORCE$^+$ manually.

## Further Systems

Some further systems for graph transformation are:

**AGG, AGG2.0.** AGG (attributed graph grammar system) is a development environment for attributed graph transformation systems with an extensible architecture. It aims at specifying and rapid prototyping of applications dealing with complex, graph structured data [Tae04]. As of AGG 2.0, it supports the creation of inverse, concurrent, amalgamated and minimal rules, as well as critical pair analysis [RET11]. The latter are not supported by ENFORCE$^+$ as of yet since its focus is on ensuring correctness of graph programs and performing graph repair.

**Progres.** PROGRES is a hybrid visual/textual language that supports programming with graph rewriting systems and also comes with a development environment. It is based on attributed graph transformation, rule oriented and imperative in nature and supports type checking, compiling and debugging graph transformation specifications via its internal tools [SWZ96].

**Groove.** GROOVE, which started as a state space generation tool, is a graph transformation tool of which some supported features are modelling via attributed typed graphs, regular expressions, priority control of transformation rules and model checking via LTL- and CTL-formulas [GdMR$^+$12]. As far as we know, graph repair is not supported in GROOVE.

**Java-Graph.** Java-Graph is a programming library for rapid development of graph tools currently in alpha stage, featuring graphs and morphisms, basic categorical constructions and application of graph transformation steps. It also offers a graphical interface for visualization and manipulation of graphs [BKM$^+$20]. As far as we know, no support for graph repair is included in the library as of yet.

## 6.5 Conclusion

In this chapter, we have summarised the capabilities of our graph repair tool ENFORCE$^{+}$ and shown how graph repair has been implemented in it. We have shown that our system generates not only equivalent, but also structurally almost equal repair programs to those obtained by hand according to the theory in all considered cases. The repair programs created by ENFORCE$^{+}$ are more or less equal to the repair programs created by hand. The small differences occur because - up to now - there is no component for optimizations included in the system. In the construction per hand, we use some optimizations rules, which are done by hand. Hence, simple generation of repair programs for graph conditions is a core feature of ENFORCE$^{+}$.

**Further topics** .

1. **Extension of ENFORCE$^{+}$ to typed graphs.**

2. **Representation of the interfaces in the output.** e.g., by marking the images of the interface morphisms.

3. **Optimization of the repair programs in the output.** The programs can be further optimized, by checking for isomorphic rules and further simplification of the application conditions, e.g., as in Nassar et al. [NKAT20].

# Chapter 7

# Conclusion

In this chapter, we summarize the results of the thesis and mention some further work.

## 7.1 Summary

In this thesis, we have presented approaches to graph generation and graph repair and an application to meta-modeling.

**Graph generation.** The main construction is a backward construction to instance generation, yielding an automaton with constraints as states. The construction is based on the construction of existential weakest liberal preconditions and constraint-guaranteeing rules. The backward construction works for arbitrary graph grammars and arbitrary constraints. The automata are closed with respect to Boolean operations complementation and product construction. The results can be easily generalized to typed attributed graph grammars. The construction terminates for specific graph grammars and graph constraints.

**Graph repair.** We have presented the theory of repair programs. The repair programs are derived directly from the given condition. We have shown that there are repair programs for legit conditions. The repair programs are stable, maximally preserving, and terminating. Based on the repair program for legit conditions, there is a grammar-based repair program for legit conditions, provided that the given rule set is compatible with the repairing sets of the original program.

**Application to meta-modeling.** Application of the repair results to the EMF-world yielded model repair for EMF constraints.

## 7.2 Further work

(1) **Generalization to type graphs,** e.g., type graph with inheritance [BET12] and arbitrary multiplicities [Tae12].

(2) **Generalization to a larger class of conditions.** Currently, we do not have a solid (ad hoc) repair program for all (satisfiable) conditions. The problem is, that not all conjunctions are preserving. Is there a solution, to get a solid (ad hoc) repair program?

(3) **Generalization of repair program with preconditions.** Similar to other notions of correct programs, the investigation of repair programs with precondition $c$ and postcondition $d$, where the correctness of the repair programs is only guaranteed for graphs, satisfying $c$.

(4) **Graph repair allowing redirection of edges.** Our repair programs try to preserve items; if this is not possible, they delete items. In Nassar et al. [NRA17], multiplicity constraints are considered. In this context, they use the idea, to redirect an edge instead of deleting it. How can the ideas of preserving graph repair and repair by redirection of edges be combined?

(5) **Complexity.** The construction of repair programs is based on the construction of repairing sets. For the repairing set of a morphism $a\colon A \hookrightarrow C$ we have to consider (at most) all subgraphs of the graph $C$. This gives an impression on the complexity for constructing the repair program of a constraint.

(6) **Implementation.** The implementation works on directed, labelled graphs. It would be important to extend the implementation to typed graphs. For better understanding of the output, the interfaces of the rules should be marked and the repair programs should be equipped with an optimization component.

(7) **Monadic second-order constraints.** A generalization of the repair programs to monadic second-order constraints (see, e.g., Courcelle [Cou97], Poskitt and Plump [PP14]), or at least path constraints [OL10] in the sense of a reflexive, transitive closure. Monadic second-order logic is more expressive than first-order logic, and many properties for practical applications require path expressions. For example, a common assumption in EMF is that there is a root node, which directly or transitively contains every other node, such that models can be edited in a tree-based editor [NRA17, Nas20].

# Appendix A

# Categories

In this appendix, we give a short introduction of the categorical terms used in this thesis. We introduce categories, show how to construct them, introduce $\mathcal{E}'$-$\mathcal{M}$ pair factorization, and present some basic constructions such as pushouts. The presentation is oriented at Ehrig et al. [EEPT06a].

In general, a category is a mathematical structure that has objects and morphisms, with a composition operation on the morphisms and an identity morphism for each object.

**Definition 33 (category).** A category $\mathbf{C} = (Ob_C, Mor_C, \circ, id)$ is defined by

- a class $Ob_C$ of objects,
- for each pair of objects $A, B \in Ob_C$, a set $Mor_C(A, B)$ of morphisms,
- for all objects $A, B, C \in Ob_C$, a composition operation $\circ_{(A,B,C)} \colon Mor_C(B, C) \times Mor_C(A, B) \to Mor_C(A, C)$, and
- for each object $A \in Ob_C$, an identity morphism $id_A \in Mor_C(A, A)$,

such that the following conditions hold:

1. Associativity. For all objects $A, B, C, D \in Ob_C$ and morphisms $f \colon A \to B$, $g \colon B \to C, h \colon C \to D$, it holds that $(h \circ g) \circ f = h \circ (g \circ f)$,
2. Identity. For all objects $A, B \in Ob_C$ and morphisms $f \colon A \to B$, it holds that $f \circ id_A = f$ and $id_B \circ f = f$.

**Remark.** Instead of $f \in Mor_C(A, B)$ we write $f \colon A \to B$ and leave out the index for the composition operation $\circ$, since it is clear which one to use. For such morphisms $f$, $A$ is called domain and $B$ is called codomain.

There are various ways to construct new categories from given existing ones. The first way is the Cartesian product of two categories, which is defined by the Cartesian product of the class of objects and the set of morphisms with componentwise compositions and identities.

**Definition 34 (product category).** Given two categories $\mathbf{C}$ and $\mathbf{D}$, the product category $\mathbf{C} \times \mathbf{D}$ is defined by

- $Ob_{C\times D} = Ob_C \times Ob_D$,
- $Mor_{C\times D}((A,A'),(B,B')) = Mor_C(A,B) \times Mor_D(A',B')$,
- for morphisms $f\colon A \to X, g\colon B \to X \in Mor_C$ and $f'\colon A' \to X', g'\colon B' \to X' \in Mor_D$, we define $(g,g') \circ (f,f') = (g \circ f, g' \circ f')$,
- $id_{(A,A')} = (id_A, id_{A'})$.

Another construction is that of a slice category. Here, the objects are morphisms of a category $\mathbf{C}$, to a distinguished object $X$. The morphisms are morphisms in $\mathbf{C}$ that connect the object morphisms so as to lead to commutative diagrams.

**Definition 35 (slice category).** Given a category $\mathbf{C}$ and an object $X \in Ob_C$, then the slice category $\mathbf{C} \setminus X$ is defined as follows:

- $Ob_{C\setminus X} = \{f\colon A \to X \mid A \in Ob_C, f \in Mor_C(A,X)\}$,
- $Mor_{C\setminus X}(f\colon A \to X, g\colon B \to X) = \{m\colon A \to B \mid g \circ m = f\}$,
- for morphisms $m \in Mor_{C\setminus X}(f\colon A \to X, g\colon B \to X)$ and $n \in Mor_{C\setminus X}(g\colon B \to X, h\colon C \to X)$, we have $n \circ m$ as defined in $\mathbf{C}$ for $m\colon A \to B$ and $n\colon B \to C$:

$$\begin{tikzcd}
A \arrow[r, "m", "="'] \arrow[dr, "f"'] & B \arrow[r, "n", "="'] \arrow[d, "g"'] & C \arrow[dl, "h"] \\
& X &
\end{tikzcd}$$

- $id_{f\colon A\to X} = id_A \in Mor_C$.

Intuitively, a pushout is an object that emerges from gluing two objects along a common subobject.

**Definition 36 (pushout).** Given morphism $f\colon A \to B$ and $g\colon A \to C \in Mor_C$, a pushout $(D, f', g')$ over $f$ and $g$ is defined by

- a pushout object $D$ and

– morphisms $f' \colon C \to D$ and $g' \colon B \to D$ with $f' \circ g = g' \circ f$,

such that the following universal property is fulfilled: for all objects $X$ with morphisms $h \colon B \to X$ and $k \colon C \to X$ with $k \circ g = h \circ f$, there is a unique morphism $x \colon D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & B \arrow[d, "g'"] \arrow[ddr, "h"] & \\
C \arrow[r, "f'"] \arrow[drr, "k"'] & D \arrow[dr, "x"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & \\
& & X
\end{tikzcd}$$

Binary coproducts can be seen as a generalization of the disjoint union of sets and graphs in a categorical framework.

**Definition 37 (binary coproduct).** Given two objects $A, B \in Ob_C$, the binary coproduct $(A + B, i_A, i_B)$ is given by

- a coproduct object $A + B$ and
- morphism $i_A \colon A \to A + B$ and $i_B \colon B \to A + B$,

such that the following universal property is fulfilled: for all objects $X$ with morphisms $f \colon A \to X$ and $g \colon B \to X$, there is a morphism $[f, g] \colon A + B \to X$ such that $[f, g] \circ i_A = f$ and $[f, g] \circ i_B = g$:

$$\begin{tikzcd}
A \arrow[r, "i_A"] \arrow[dr, "f"'] \arrow[dr, phantom, "=", bend left] & A + B \arrow[d, "{[f,g]}"] \arrow[d, phantom, "=", shift left=4] & B \arrow[l, "i_B"'] \arrow[dl, "g"] \\
& X &
\end{tikzcd}$$

**Definition 38 (jointly epimorphic).** A morphism pair $(e_1, e_2)$ with $e_i \colon A_i \to B$ $(i = 1, 2)$ is called *jointly epimorphic* if, for all $g, h \colon B \to C$ with $g \circ e_i = h \circ e_i$ for $i = 1, 2$, we have $g = h$.

We use the $\mathcal{E}'$-$\mathcal{M}$ pair factorization for the Shift construction (see Lemma 3).

**Definition 39 ($\mathcal{E}'$-$\mathcal{M}$ pair factorization).** Given a class of morphism pairs $\mathcal{E}'$ with the same codomain, a (weak) adhesive high-level replacement category $\mathcal{C}$ has an $\mathcal{E}'$-$\mathcal{M}$ *pair factorization* if, for each pair of morphisms $f_1\colon A_1 \to C$ and $f_2\colon A_2 \to C$, there exists an object $K$ and morphisms $e_1\colon A_1 \to K, e_2\colon A_2 \to K$, and $m\colon K \to C$ with $(e_1, e_2) \in \mathcal{E}'$ and $m \in \mathcal{M}$ such that $m{\circ}e_1 = f_1$ and $m{\circ}e_2 = f_2$.

$$\begin{tikzcd}
A_1 \arrow[dr, "e_1"] \arrow[drr, bend left, "f_1"] & = & \\
& K \arrow[r, "m"'] & C \\
A_2 \arrow[ur, "e_2"'] \arrow[urr, bend right, "f_2"'] & = &
\end{tikzcd}$$

The idea of a van Kampen square is that of a pushout which is stable under pullbacks, and, vice versa, where pullbacks are stable under combined pushouts and pullbacks.

**Definition 40 (van Kampen square).** A pushout (1) is a van Kampen square if, for any commutative cube (2) with (1) in the bottom and where the back faces are pullbacks, the following statement holds: the top face is a pushout iff the front faces are pullbacks.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "g'"] \\
C \arrow[r, "f'"] & D
\arrow[from=1-1, to=2-2, phantom, "(1)"]
\end{tikzcd}$$

$$\begin{tikzcd}
& & A' \arrow[dll, "f'"'] \arrow[dr, "m'"] \arrow[dd, "a"] & \\
C' \arrow[dr, "n'"] \arrow[dd, "c"] & & & B' \arrow[dll, "g'"] \arrow[dd, "b"] \\
& D' \arrow[dd, "d"] & A \arrow[dll, "f"'] \arrow[dr, "m"] & \\
C \arrow[dr, "n"'] & & & B \arrow[dll, "g"] \\
(2) & D & &
\end{tikzcd}$$

**Definition 41 ($\mathcal{M}$-adhesive category).** A category **C** with a morphism class $\mathcal{M}$ is called an $\mathcal{M}$-*adhesive category* if

- $\mathcal{M}$ is a class of monomorphisms closed under isomorphism, composition ($f\colon A \to B \in \mathcal{M}, g\colon B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M}$), and decomposition ($g \circ f \in \mathcal{M}, g \in \mathcal{M} \Rightarrow f \in \mathcal{M}$).
- **C** has pushouts and pullbacks along $\mathcal{M}$-morphisms, and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks.
- Pushouts in **C** along $\mathcal{M}$-morphisms are $\mathcal{M}$-van Kampen squares.

**Lemma 30.** $\mathbf{Graphs_{TG}}$ is $\mathcal{M}$-adhesive and has $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

**Proof.** *$\mathcal{M}$-adhesiveness.* The proof is done by constructing a slice category to typed graphs (without containment). The category of typed graph with containment, is isomorphic to the slice category to typed graphs (without containment). Let $TG = (T, C, O)$, with $T = (V, E, s, t)$ be a type graph. Consider the slice category $\mathbf{Graphs_{TG}} \backslash O$. Each typed graph in this slice category is isomorphic to the graph in $\mathbf{Graphs_{TG}}$, where each edge, $e \in O$ is in $E$, and the typed graph morphisms are exactly the morphisms in this slice category. Consider a graph $LTG$, i.e., a labelled type graph, where each (typed) containment edge $e \in C$ is represented by a (typed) labelled edge with the special label symbol for the containment edges, i.e., $l_E(e) =$ ◆. By Ehrig et al. [EEPT06a] labelled graphs form a category $\mathbf{Graphs_{LG}}$. By [EEPT06a, Theorem 4.15], the product category $\mathbf{Graphs_{LTG}}$ is a category.

By [EEPT06a, Fact 4.16], typed graphs (without containment) are adhesive categories and, by the hierarchy of adhesive categories in [EGH10], $\mathcal{M}$-adhesive. By [EEPT06a, Theorem 4.15], if $\mathcal{C}$ is a category, so is the slice category. Consequently, typed graphs with containment are an $\mathcal{M}$-adhesive category.

*$\mathcal{E}'$-$\mathcal{M}$ pair factorization.* By [EEPT06a, Remark 5.26], The intuitive idea of morphism pairs $(e_1, e_2) \in \mathcal{E}'$ is that of jointly epimorphic morphisms (see Definition 38). This can be established in categories with binary coproducts and an $\mathcal{E}_0$-$\mathcal{M}_0$ pair factorization of morphisms, where $\mathcal{E}_0$ is a class of epimorphisms and $\mathcal{M}_0$ a class of monomorphisms. *Binary coproducts*: By Ehrig et al. [EEPT06a, Example A.28], the category of typed graphs, and the category of labelled graphs have binary coproducts, and, in a product or slice category, coproducts can be constructed componentwise if the underlying categories have coproducts. Since the category of typed graphs (with containment) is constructed as a product category (see Definition 34), it has binary coproducts. $\mathcal{E}_0$-$\mathcal{M}_0$ *pair factorization*: Given $A_1 \rightarrow^{f_1} C \leftarrow^{f_2} A_2$, we take an $\mathcal{E}_0$-$\mathcal{M}_0$ pair factorization $f = m \circ e$ of the induced morphism $f \colon A_1 + A_2 \to C$ and define $e_1 = e \circ \iota_1$ and $e_2 = e \circ \iota_2$, where $\iota_1, \iota_2$ are the coproduct injections:

$$\begin{tikzcd}
A_1 \arrow[rr, "\iota_1"] \arrow[drrr, dotted, "e_1"'] \arrow[ddrr, "f_1"'] & & A_1 + A_2 \arrow[dr, "e"] \arrow[dd, "f"'] & & A_2 \arrow[ll, "\iota_2"'] \arrow[dl, dotted, "e_2"] \arrow[ddll, "f_2"] \\
& & & K \arrow[dl, "m"'] & \\
& & C & &
\end{tikzcd}$$

An $\mathcal{E}_0$-$\mathcal{M}_0$ pair factorization in the category of typed graphs (with containment) is given by the classes $\mathcal{E}_0$ of surjective morphisms and $\mathcal{M}_0$ of injective morphisms. Let $f_i\colon A_i \to C$ with typing $type_i\colon A_i \to LTG, type\colon C \to LTG$. Compute the $\mathcal{E}'$-$\mathcal{M}$ pair factorization in the category of labelled $\mathbf{Graphs_{LG}}$ - ignoring the typing - and choose the typing morphism $type_K = type_C \circ m$, i.e., $type_K(x) = type_C(m(x))$ for $x \in K$, The morphisms $e_1, e_2$ respect the typing since $type_K(e_i(x)) = type_C(f_i(x)) = type_i(x)$. □

# Bibliography

[AHPZ06] Karl Azab, Annegret Habel, Karl-Heinz Pennemann, and Christian Zuckschwerdt. ENFORCe: A system for ensuring formal correctness of high-level programs. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 1, 2006.

[AHRT14] Thorsten Arendt, Annegret Habel, Hendrik Radke, and Gabriele Taentzer. From core OCL invariants to nested graph constraints. In *Graph Transformations (ICGT 2014)*, volume 8571 of *Lecture Notes in Computer Science*, pages 97–112, 2014.

[AHS90] Jiří Adámek, Horst Herrlich, and George Strecker. *Abstract and Concrete Categories*. John Wiley, 1990.

[AJ01] Parosh Aziz Abdulla and Bengt Jonsson. Ensuring completeness of symbolic verification methods for infinite-state systems. *TCS*, 256(1-2):145–167, 2001.

[AM75] Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors*. Academic Press, 1975.

[AO91] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts and Monographs in Computer Science. Springer, 1991.

[BDK$^+$12] Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier, and Jan Stückrath. On the decidability status of reachability and coverability in graph transformation systems. In *RTA*, volume 15 of *LIPIcs*, pages 101–116, 2012. Long version: Technical Report DISI-TR-11-04, Università di Genova, 2012.

[Bec16] Jan Steffen Becker. An automata-theoretic approach to instance generation. In *Graph Computation Models (GCM 2016), Electronic Pre-Proceedings*, 2016.

[Ber14] Gábor Bergmann. Translating OCL to graph patterns. In *Model-Driven Engineering Languages and Systems (MODELS 2014)*, LNCS, pages 670–686, 2014.

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

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *16th IEEE International Conference on Automated Software Engineering*, pages 273–280. IEEE Computer Society, 2001.

[BHH12] Gábor Bergmann, Dóra Horváth, and Ákos Horváth. Applying incremental graph transformation to existing models in relational databases. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations - 6th International Conference, ICGT*, volume 7562 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2012.

[BKM$^+$20] H. J. Sander Bruggink, Barbara König, Marleen Matjeka, Dennis Nolte, and Lars Stoltenow. A flexible and easy-to-use library for the rapid development of graph tools in java. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF*, volume 12150 of *Lecture Notes in Computer Science*, pages 297–306. Springer, 2020.

[BMST99] Roswitha Bardohl, Mark Minas, Andy Schürr, and Gabriele Taentzer. Application of graph transformation to visual languages. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 105–180. World Scientific, 1999.

[BRH19] Angela Barriga, Adrian Rutle, and Rogardt Heldal. Personalized and automatic model repairing using reinforcement learning. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion*, pages 175–181. IEEE, 2019.

[CCR07] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 547–548, 2007.

[CCYW18] Yurong Cheng, Lei Chen, Ye Yuan, and Guoren Wang. Rule-based graph repairing: Semantic and efficient repairing methods. In *34th IEEE International Conference on Data Engineering, ICDE 2018,*, pages 773–784, 2018.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, pages 313–400. World Scientific, 1997.

[DHM20] Frank Drewes, Berthold Hoffmann, and Mark Minas. Graph parsing as graph transformation - correctness of predictive top-down parsers. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT*, volume 12150 of *Lecture Notes in Computer Science*, pages 221–238. Springer, 2020.

[Din92] Guoli Ding. Subgraphs and well-quasi-ordering. *Journal of Graph Theory*, 16(5):489–502, 1992.

[dLBE+07] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007.

[EEdL+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.

[EEHP06] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundamenta Informaticae*, 74(1):135–166, 2006.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[EEPT06a] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory of typed attributed graph transformation based on adhesive HLR categories. *Fundamenta Informaticae*, 74(1):31–61, 2006.

[EEPT06b] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs of Theoretical Computer Science. Springer, 2006.

[EGH10] Hartmut Ehrig, Ulrike Golas, and Frank Hermann. Categorical frameworks for graph transformation and HLR systems based on the DPO approach. *Bulletin of the EATCS*, 112:111–121, 2010.

[EGH$^+$14] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: Parallelism, concurrency and amalgamation. *Mathematical Structures in Computer Science*, 24, 2014.

[EHK$^+$97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation. Part II: Single-pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, pages 247–312. World Scientific, 1997.

[EHKP91] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski, and Francesco Parisi-Presicce. Parallelism and concurrency in high level replacement systems. *Mathematical Structures in Computer Science*, 1:361–404, 1991.

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69, 1979.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph grammars: An algebraic approach. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180, Iowa City, 1973.

[Fli16] Nils Erik Flick. *Correctness of Structure-Changing Systems under Adverse Conditions*. PhD thesis, Universität Oldenburg, 2016. `http://oops.uni-oldenburg.de/2895`.

[GdMR+12] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *Int. J. Softw. Tools Technol. Transf.*, 14(1):15–40, 2012.

[GLEO12] Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. *Theor. Comput. Sci.*, 424:46–68, 2012.

[Gro03] Object Management Group. Mda guide, 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.

[HLBG12] Stephan Hildebrandt, Leen Lambers, Basil Becker, and Holger Giese. Integration of triple graph grammars and constraints. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 54, 2012.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245, 2001.

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

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Graph Transformations (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460, 2006.

[HS18] Annegret Habel and Christian Sandmann. Graph repair by graph programs. In *Graph Computation Models (GCM 2018)*, volume 11176 of *Lecture Notes in Computer Science*, pages 431–446, 2018.

[HST18] Annegret Habel, Christian Sandmann, and Tilman Teusch. Integration of graph constraints into graph grammars. In *Graph Transformation, Specifications, and Nets*, volume 10800 of *Lecture Notes in Computer Science*, pages 19–36, 2018.

[HT20] Reiko Heckel and Gabriele Taentzer. *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer, 2020.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Compuation*. Addison-Wesley, 1979.

[Hub21] Marius Hubatschek. Graph repair of multiplicity constraints. Master thesis, University of Oldenburg, 2021. Submitted.

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars — a constructive approach. In *SEGRAGRA '95*, volume 2 of *Electronic Notes in Theoretical Computer Science*, pages 95–104, 1995.

[Jac12] Daniel Jackson. Alloy Analyzer website, 2012. `http://alloy.mit.edu/`.

[Jeu18] Manfred A. Jeusfeld. Metamodel. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. `https://doi.org/10.1007/978-1-4614-8265-9_898`.

[KG12] Mirco Kuhlmann and Martin Gogolla. From UML and OCL to Relational Logic and Back. In *Model Driven Engineering Languages and Systems (MODELS 2012)*, volume 7590 of *LNCS*, pages 415–431, 2012.

[KLT07] Christian Köhler, Holger Lewin, and Gabriele Taentzer. Ensuring containment constraints in graph-based model transformation approaches. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 6, 2007.

[KSTZ20] Jens Kosiol, Daniel Strüber, Gabriele Taentzer, and Steffen Zschaler. Graph consistency as a graduated property - consistency-sustaining and -improving graph transformations. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT*, volume 12150 of *Lecture Notes in Computer Science*, pages 239–256. Springer, 2020.

[LKSS15] Michael Löwe, Harald König, Christoph Schulz, and Marius Schultchen. Algebraic graph transformations with inheritance and abstraction. *Sci. Comput. Program.*, 107-108:2–18, 2015.

[Löw93] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.

[LS05] Stephen Lack and Paweł Sobociński. Adhesive and quasiadhesive categories. *Theoretical Informatics and Application*, 39(2):511–546, 2005.

[MKW20] Johannes Meier, Ruthbetha Kateule, and Andreas Winter. Operator-based viewpoint definition. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, pages 401–408. SCITEPRESS, 2020.

[MTC17] Nuno Macedo, Jorge Tiago, and Alcino Cunha. A feature-based classification of model repair approaches. *IEEE Trans. Software Eng.*, 43(7):615–640, 2017.

[MW18] Johannes Meier and Andreas Winter. Model consistency ensured by metamodel integration. In *Proceedings of MODELS*, volume 2245 of *CEUR Workshop Proceedings*, pages 408–415. CEUR-WS.org, 2018.

[Nas20] Nebras Nassar. *Consistency-by-Construction Techniques for Software Models and Model Transformations.* PhD thesis, University Marburg, 2020.

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

[NKAT20] Nebras Nassar, Jens Kosiol, Thorsten Arendt, and Gabriele Taentzer. Constructing optimized constraint-preserving application conditions for model transformation rules. *J. Log. Algebraic Methods Program.*, 114:100564, 2020.

[NKR17] Nebras Nassar, Jens Kosiol, and Hendrik Radke. Rule-based repair of emf models: Formalization and correctness proof. In *Graph Computation Models (GCM 2017)*, 2017. `http://pages.di.unipi.it/corradini/Workshops/GCM2017/papers/Nassar-Kosiol-Radke-GCM2017.pdf`.

[NRA17] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. Rule-based repair of EMF models: An automated interactive approach. In *Theory and Practice of Model Transformation (ICMT 2017)*, volume 10374 of *Lecture Notes in Computer Science*, pages 171–181, 2017.

[Obj14] Object Management Group. Object constraint language, version 2.4, OCL (february 2014). https://www.omg.org/spec/OCL/2.4/, 2014.

[OL10] Fernando Orejas and Leen Lambers. Symbolic attributed graphs for attributed graph transformation. *Electronic Communications of the EASST*, 30, 2010.

[OPNL18] Fernando Orejas, Elvira Pino, Marisa Navarro, and Leen Lambers. Institutions for navigational logics for graphical structures. *Theor. Comput. Sci.*, 741:19–24, 2018.

[Pen04] Karl-Heinz Pennemann. Generalized constraints and application conditions for graph transformation systems. Diploma thesis, University of Oldenburg, 2004.

[Pen08] Karl-Heinz Pennemann. An algorithm for approximating the satisfiability problem of high-level conditions. In *Proc. Int. Workshop on Graph Transformation for Verification and Concurrency (GT-VC'07)*, volume 213 of *Electronic Notes in Theoretical Computer Science*, pages 75–94, 2008.

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

[Plu05] Detlef Plump. Confluence of graph transformation revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, volume 3838 of *Lecture Notes in Computer Science*, pages 280–308, 2005.

[PP13] Christopher M. Poskitt and Detlef Plump. Verifying total correctness of graph programs. *Electronic Communications of the EASST*, 61, 2013.

[PP14] Christopher M. Poskitt and Detlef Plump. Verifying monadic second-order properties of graph programs. In *Graph Transformation (ICGT 2014)*, volume 8571 of *Lecture Notes in Computer Science*, pages 33–48, 2014.

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

[RAB$^+$15] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Grabriele Taentzer. Translating essential ocl invariants to nested graph constraints focusing on set operations. In *Graph Transformation (ICGT 2015)*, volume 155-170 of *Lecture Notes in Computer Science*, page 9151, 2015.

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

[Rad16] Hendrik Radke. *A Theory of HR$^*$ Graph Conditions and their Application to Meta Modeling*. PhD thesis, Universität Oldenburg, 2016.

[Ren04] Arend Rensink. Representing first-order logic by graphs. In *Graph Transformations (ICGT'04)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335, 2004.

[RET11] Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 - new features for specifying and analyzing algebraic graph transformations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Applications of Graph Transformations with Industrial Relevance - 4th International Symposium, AGTIVE*, volume 7233 of *Lecture Notes in Computer Science*, pages 81–88. Springer, 2011.

[RLYK15] Fazle Rabbi, Yngve Lamo, Ingrid Chieh Yu, and Lars Michael Kristensen. A diagrammatic approach to model completion. In *Analysis of Model Transformations*, volume 1500 of *CEUR Workshop Proceedings*, pages 56–65. CEUR-WS.org, 2015.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[Rut10] Adrian Rutle. *Diagram Predicate Framework*. PhD thesis, University of Bergen, 2010.

[Sag19] Jens Sager. A modeling framework with model repair by graph programs. Master thesis, University of Oldenburg, 2019. `https://uol.de/f/2/dept/informatik/ag/fs/Sager19.pdf`.

[San20] Christian Sandmann. Graph repair and its application to metamodelling. In *Graph Computation Models (GCM 2020)*, volume 330 of *Electronic Proceedings in Theoretical Computer Science*, pages 13 – 34. Open Publishing Association, 2020.

[SBG+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A usability-focused framework for EMF model transformation development. In *Graph Transformation - 10th International Conference, ICGT*, volume 10373 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2017.

[SBL+20] Oszkár Semeráth, Aren A. Babikian, Anqi Li, Kristóf Marussy, and Dániel Varró. Automated generation of consistent models with structural and attribute constraints. In Eugene Syriani, Houari A. Sahraoui, Juan de Lara, and Silvia Abrahão, editors, *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*, pages 187–199. ACM, 2020.

[SBMP08] David Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional, 2008.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, Proceedings*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[SH19] Christian Sandmann and Annegret Habel. Rule-based graph repair. In *Proceedings Tenth International Workshop on Graph Computation Models, GCM@STAF 2019*, volume 309 of *Electronic Proceedings in Theoretical Computer Science*, pages 87–104, 2019.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[SLO17] Sven Schneider, Leen Lambers, and Fernando Orejas. Symbolic model generation for graph properties. In *Fundamental Approaches to Software Engineering (FASE 2017)*, volume 10202 of *Lecture Notes in Computer Science*, pages 226–243, 2017.

[SLO18] Sven Schneider, Leen Lambers, and Fernando Orejas. Automated reasoning for attributed graph properties. *Int. J. Softw. Tools Technol. Transf.*, 20(6):705–737, 2018.

[SLO19] Sven Schneider, Leen Lambers, and Fernando Orejas. A logic-based incremental approach to graph repair. In *Fundamental Approaches to Software Engineering - (FASE 2019)*, volume 11424 of *Lecture Notes in Computer Science*, pages 151–167, 2019.

[SNV18] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. A graph solver for the automated generation of consistent domain-specific models. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE*, pages 969–980. ACM, 2018.

[SWZ96] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Developing tools with the PROGRES environment. In Manfred Nagl, editor, *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*, pages 356–369. Springer, 1996.

[Tae04] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453, 2004.

[Tae12] Gabriele Taentzer. Instance generation from type graphs with arbitrary multiplicities. *Electronic Communications of the EASST*, 47, 2012.

[TOLR17] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. Change-preserving model repair. In *Fundamental Approaches to Software Engineering (ETAPS 2017)*, volume 10202 of *Lecture Notes in Computer Science*, pages 283–299, 2017.

[VFV06] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a graph transformation engine in relational databases. *Softw. Syst. Model.*, 5(3):313–341, 2006.

[VSSH18] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, volume 10800 of *Lecture Notes in Computer Science*, pages 285–312. Springer, 2018.

[Wac07] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming,* *21st European Conference*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer, 2007.

[Wan16] Xiaoliang Wang. *Towards Correct Modelling and Model Transformation in DPF*. PhD thesis, University of Bergen, 6 2016.

# List of Symbols

| *symbol* | *meaning* |
|---|---|
| $a\colon A \hookrightarrow C$ | morphism in conditions from $A$ to $C$ |
| ac | application condition |
| $ac_L$ | left application condition |
| $A$ | automaton |
| $\mathcal{A}$ | constraint automaton |
| $c, d$ | conditions |
| $C$ | set of containment edges |
| $[\![c]\!]$ | class of all graphs satisfying $c$. |
| $ds = d_1, \ldots, d_n$ | sequence of conditions |
| $d_i$ | condition |
| $\Rightarrow$ | transformation |
| $\delta(g, d)$ | maximal number of necessary deletions |
| $e_1$ | conjunction of negative conditions |
| $e_2$ | conjunction of universal conditions |
| $E_G$ | set of edges of a graph $G$ |
| EMF constraints | At most one container, no containment cycles, no parallel edges, all opposite edges |
| EMF$k$ constraints | first-order version of $EMF$k constraints |
| EMF-model graph | graph satisfying $EMF$ constraints |
| emf | conjunction of EMF constraints |
| emf$k$ | conjunction of EMF$k$ constraints |
| $\mathrm{Ext}(g)$ | extended morphisms of $g$ |
| $g\colon A \hookrightarrow G$ | morphism from $A$ to $G$ |
| $g'\colon L \hookrightarrow G$ | morphism from $L$ to $G$ |
| $G$ | (input) graph |
| $GG$ | graph grammar |
| $GG$-based program | program consisting of rules of a grammar $GG$ |
| $\mathbf{Graphs_{TG}}$ | category of typed graphs |
| $\mathrm{Gua}(\varrho, d)$ | $d$-guaranteeing application condition |

| | |
|---|---|
| $\mathrm{gua}(\varrho, d) = \langle \varrho, \mathrm{Gua}(\varrho, d) \rangle$ | $d$-guaranteeing rule |
| $h\colon A \hookrightarrow H$ | morphism from $A$ to $H$ |
| $h'\colon R \hookrightarrow H$ | morphism from $R$ to $H$ |
| $H$ | (resulting) graph |
| $i$ | interface relation |
| $\hookrightarrow$ | injective morphism |
| $L$ | left-hand side of a rule |
| $L(GG)$ | language of a graph grammar |
| $L(GG) \cap \llbracket c \rrbracket$ | language of a graph grammar satisfying constraint $c$ |
| Left | shift over rules (from right to left) |
| $\mathcal{N}$ | set of nonterminal symbols |
| $n$-bounded graph | path in graph has length $\leq n$ |
| $n$-bounded graph grammar | all generated graph by grammar are $n$-bounded |
| $O$ | relation of opposite edges |
| $p$ | plain rule |
| $\varrho$ | rule (with interface and application condition) |
| $\varrho^k$ | rule equipped with context |
| $P$ | program (with interface) |
| $Ps = P_1, \dots P_n$ | sequence of programs |
| $P_d$ | solid repair program for $d$ |
| $P^d$ | $d$-preserving program of $P$ |
| $\mathrm{Pres}(\varrho, d)$ | $d$-preserving application condition |
| $Q_1$ | repair program for $e_1$ |
| $Q_2$ | repair program for $e_2$ |
| $Q_2^{\prime e_1}$ | $e_1$-preserving repair program for $e_2$ |
| $R$ | right-hand side of a rule |
| $\mathcal{R}$ | rule set |
| $\mathcal{R}_a$ | increasing repairing set for $\exists\, a$ |
| $S$ | start graph of a graph grammar |
| $\mathcal{S}_a$ | decreasing repairing set for $\nexists\, a$ |
| Shift | shift over morphisms |
| $\sqsubseteq$ | containment of conditions |
| $t$ | transformation |
| $TG$ | type graph |
| tr | track morphism |
| $type_G\colon G \to T$ | typing morphism |
| $V_G$ | set of nodes in a graph $G$ |
| $\mathrm{Wlp}_\exists$ | construction for existential weakest liberal precondition |
| $x, y$ | left and right interface morphisms |

# Index

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe. Ich versichere, dass ich im Zusammenhang mit dem Promotionsvorhaben keine kommerziellen Vermittlungs- oder Beratungsdienste (Promotionsberatung) in Anspruch genommen habe.

I hereby confirm that I completed the work independently and used only the indicated resources. Furthermore, I confirm that I am aware of the guidelines of good scientific practice of the Carl von Ossietzky University Oldenburg and that I observed them while writing this dissertation. I did not use any commercial dispatching or counseling services concerning the dissertation.

Unterschrift