CARL VON OSSIETZKY universität OLDENBURG

Department für Informatik
Abteilung Formale Sprachen
Master Informatik

Masterarbeit

A Modeling Framework with Model Repair by Graph Programs

Vorgelegt von:	Jens Sager 4430077 Jens.Sager@posteo.de
Betreuende Gutachterin:	Prof. Dr. Annegret Habel
Zweiter Gutachter:	Christian Sandmann
Abgabe:	05. September 2019

Abstract

The thesis introduces a meta-modeling framework with formal constraint semantics and the ability to repair models by existing graph repair approaches. It is based on the Diagram Predicate Framework of Rutle 2010 and uses graph conditions in the sense of Habel and Pennemann 2009.

Model equivalence is undecidable in general. We define constructions for the equivalent transformation of models and show their correctness. This allows the application of existing graph repair approaches. We use this to define a repair procedure that is proven correct for models with a preserving execution order of constraint repair programs. Finally, we show that it is undecidable in general whether a given graph program is preserving or guaranteeing for a constraint.

Contents

1. Introduction

1.1. Motivation

This section lays out the motivation and the reasoning for the overarching choices made in this thesis.

Model Driven Engineering

Models see widespread use in the design of both physical and software systems. A model is a simplified representation of reality. For example, some real world systems can be modeled using the laws of physics and establishing a set of differential equations describing their behaviour. A software system may be modeled by abstracting its components and their communication. Models are a means to better understand the system to be designed.

This process is taken one step further by approaches like model based systems engineering where the model is used to simulate a physical system on a computer allowing aspects of it to be tested without actually producing it first. On the software engineering side a sufficiently detailed model can be used to automatically generate an equivalent implementation.

Various specification languages exist for different kinds of models. The Unified Modeling Language (UML) and its derivates like the Systems Modeling Language (SysML) are used for software and systems design. The Business Process Model and Notation (BPMN) is used to graphically specify business processes. The Eclipse Modeling Framework (EMF) allows code generation from its models.

Meta-modeling

A meta-model is a model that does not directly specify a system but another model. Stacking meta-models allows the design on a system on different levels of abstraction going from most abstract to most specific. This is a common technique in system design.

This approach has the advantage of making lower level models exchangeable without touching the overarching structure. It may also aid the understanding of the system to people unfamiliar with it by explaining the bigger concepts before going into detail. In a system development context this may take the form of various block diagrams where each box represents a subsystem that is then again explained in detail by its own block diagram. This can be taken to the lowest levels of abstraction and control where even the underlying physical properties

may be modeled using blocks. Examples of this would be MATLAB/Simulink and Stateflow, Scilab or Octave.

Model Repair

The modeling framework we introduce here includes steps to generate automatic repair programs for the constructed model. Why would this be necessary? Given the ability to repair any model instance it should similarly be possible to prevent mistakes in the creation of any instance to begin with, thus making repair obsolete.

The main advantage of specifying repair programs is that they can be applied to existing instances when the corresponding model changes. Additionally one may formally model a system after its instances are already established and then adapt the instance to this specification while making as little changes as possible. One example of this would be a company introducing a new management system while keeping known structures where possible to reduce the cost of retraining employees.

1.2. Goals

This thesis has two main goals:

Modeling: Define a layered modeling framework using typed graphs for its structure and graph conditions [HP09] to define model constraints. The approach is inspired by the Diagram Predicate Framework (DPF) [Rut10].

Repair: Investigate the automatic repair of model instances to conform to these constraints. Here graph programs are used for the construction of repair programs. When given any adequately typed instance graph as an input these programs shall construct a conforming instance.

Additionally the following optional goals are defined:

Repair for model families: The definition of repair programs for a generalized set of models derived from similar constraints can be investigated. The result could be general repair algorithms if a model uses certain constraint types like the approach for multiplicities in [NRA17]. A corresponding program would merely need its typing instantiated to the model at hand to be a repair program.

Identification of frequent constraints: To identify relevant model families, a set of common constraints on models in software engineering may be provided. Additionally the necessary expressiveness of a constraint formalism for the constraints can be established. This would provide guidance for a possible implementation in determining how much use a given constraint formalism has.

The remainder of this thesis is structured as follows: Chapter 2 introduces graph conditions, transformations and programs used throughout the thesis. Chapter 3 gives an overview of the diagram predicate framework. In Chapter 4 a constraint based modeling framework based on DPF is defined. Predicates are replaced with graph conditions. Definitions for models, model equivalence and valid instances are provided. Chapter 5 presents an approach to repairing these models. A construction to equivalently transform models such that existing graph repair methods can be used on them is provided. Additionally an evaluation of the new modeling framework in comparison to its basis in DPF is provided. Chapter 6 gives an overview of related work. Chapter 7 outlines the main contributions of this thesis as well as plans for future work. Appendix A gives a short introduction to category theoretic terms used in this thesis. It also shows two possible future extensions for constraints.

2. Graph Conditions and Programs

This chapter introduces graph conditions [HP09] and graph programs [HP01]. To use them for our modeling and repair tasks we also generalize them to be applicable on typed graphs.

2.1. Graphs

In this thesis, graphs are used as the central structure for modeling as well as the graph repair definitions. Graphs see widespread use and study in computer science and software development as well as engineering. One of their advantages is to be easily visualized giving an intuitive overview of an otherwise complex system. The definition of what constitutes a graph is adapted from [Rut10].

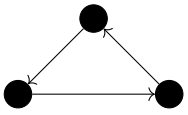
Definition 1 (graph)**.** A **graph** G is a structure

$$G = (G_0, G_1, src^G, trg^G)$$

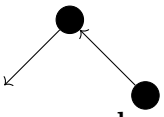
with a set of **nodes** or **vertices** G_0, a set of **edges** G_1 and total functions $src^G : G_1 \to G_0$ and $trg^G : G_1 \to G_0$ mapping the source and target nodes to each edge.

Note. The functions src and trg are required to be total. This means no edge may be without a source or target. Such an edge is called **dangling**. An additional advantage of giving edges via a set is that it allows for multiple edges between the same nodes.

Example 1. Of the two structures below the left one is a graph consisting of three nodes and edges. The right one is not a graph as it has a dangling edge with no target.



graph



no graph

In order to express relations between graphs and facilitate graph transformations maps between graphs called graph morphisms are used.

Definition 2 (graph morphism)**.** A **graph morphism** $\phi : G \to H$ between graphs G and H is a pair of functions $\phi_0 : G_0 \to H_0$, $\phi_1 : G_1 \to H_1$ preserving sources and targets on edges:

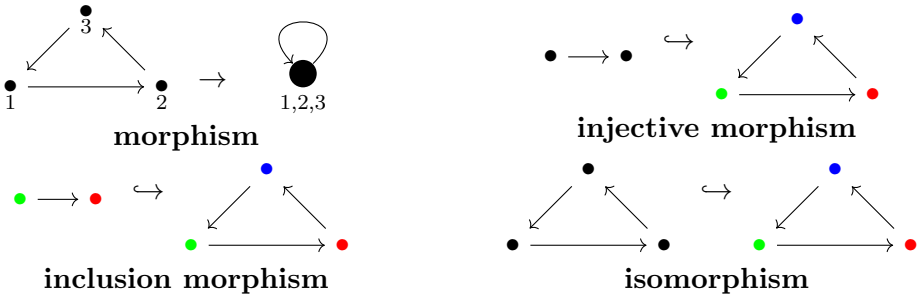
$$\begin{aligned} \forall e \in G_1 : & \phi_0(src^G(e)) = src^H(\phi_1(e)) \\ \text{and} \quad & \phi_0(trg^G(e)) = trg^H(\phi_1(e)) \end{aligned}$$

A graph morphism ϕ is **injective** (**surjective**) if both ϕ_0 and ϕ_1 are injective (surjective). An injective morphism is written with a hooked arrow $G \hookrightarrow H$. It is an **inclusion** and G a **subgraph** of H if $G_0 \subseteq H_0$ and $G_1 \subseteq H_1$ and:

$$src^G(e) = src^H(e) \text{ and}$$
$$trg^G(e) = trg^H(e) \text{ for all } e \in G_1$$

A morphism is an **isomorphism** if ϕ_0 and ϕ_1 are both injective and surjective.

Example 2. Different types of morphisms are shown below. Here two nodes get the same colour if they refer to the exact same object. When nodes are unified in a morphism they are marked with a number. The important difference between an injective morphism and an inclusion is that the inclusion demands the exact objects of the source be present in the target. For an injective morphism the occurrence of the shape suffices.



Definition 3 (typed graphs)**.** A **typed graph** (G,ι) is a graph G together with a graph morphism $\iota : G \to TG$ called a typing morphism. G is said to be **typed by** TG. TG is called a **type graph**. The typing morphism of graph G is commonly written as ι^G. When the typed graph and corresponding typing morphism of a graph are unambiguous, its elements may be shown in the shape of their type and the arrows indicating the typing morphism omitted.

A **typed graph morphism** is defined as a morphism $\phi : (G,\iota^G) \to (H,\iota^H)$ that constitutes a graph morphism $\overline{\phi} : G \to H$ such that $\iota^G = \iota^H \circ \phi$. For any typed graph morphism a the underlying untyped graph morphism is denoted as $\overline{a}$.

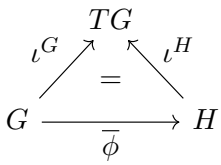


Figure 2.1.: Visualisation of a type graph morphism ϕ.

Example 3. In Fig. 2.2 a typed graph morphism ϕ is shown. The graphs G and H are typed by a common type graph TG. The types of their individual nodes are given by their corresponding shape in the type graph. The mapping of nodes by $\overline{\phi}$ is indicated by numbering the nodes of G.

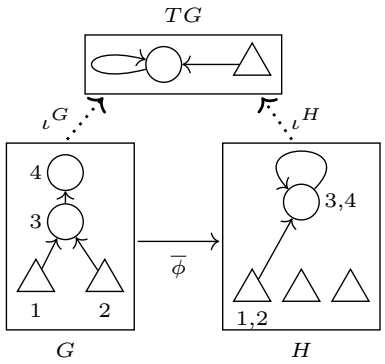


Figure 2.2.: Two typed graphs with a morphism between them.

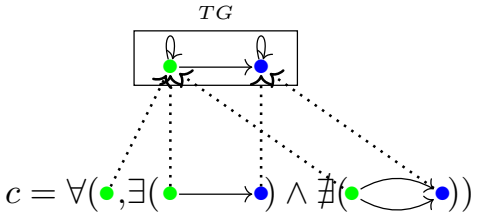
2.2. Graph Conditions

This section introduces nested graph conditions and constraints [HP09]. To apply the concept to layered typed graphs, we extend them with typing.

Definition 4 (conditions)**.** A **(typed) graph condition** over a graph P is either $true$ or $\exists(a,c)$ with (typed) inclusion morphism $a : P \hookrightarrow C$ and a (typed) condition c over C. Additionally, boolean operators are defined such that if c_1 and c_2 are conditions so are $c_1 \wedge c_2$ and $\neg c_1$.
Other boolean operators are defined as usual. When the domain of $a : P \hookrightarrow C$ can be unambiguously inferred only the codomain C is written. The following abbreviations are used:

- $\exists a \equiv \exists(a,true)$
- $\forall(a,c) \equiv \neg\exists(a,\neg c)$
- $false \equiv \neg true$

Example 4. The following typed graph condition informally requires that every green node in the graph has an outgoing edge to a blue node but not two to the same node. Note that each graph in the condition is typed by the same type graph TG as indicated by the dotted edges. The edge typing is omitted as it is unambiguous.

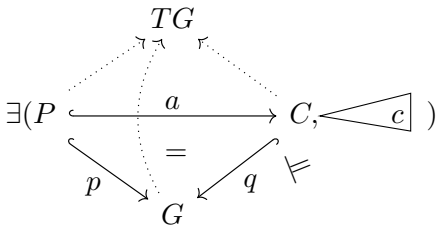


Fact 1. [Pen09] Graph conditions are equivalent in expressiveness to first-order graph formulas.

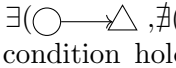
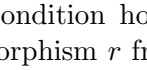
Definition 5 (semantics of conditions)**.** The semantics of (typed) graph conditions are defined inductively as follow:

- Every injective (typed graph) morphism satisfies *true*.
- An injective (typed graph) morphism $p : P \to G$ satisfies $\exists(a,c)$ if there exists an injective (typed graph) morphism $q : C \to G$ such that $q \circ a = p$ and q satisfies c.
- An injective (typed graph) morphism $p : P \to G$ satisfies $\neg c$ over P if p does not satisfy c and p satisfies $c_1 \wedge c_2$ if p satisfies both c_1 and c_2
- A graph G satisfies a condition $\exists(a,c)$ if the condition is over the empty graph and the (typed graph) morphism $p : \emptyset \to G$ satisfies the condition.

A condition c being satisfied by some morphism m is written as $m \models c$. Similarly a graph G satisfying the condition is written as $G \models c$.



Note. The difference between a typed and untyped graph condition is, that every graph in a typed condition is a typed graph while the graphs in untyped conditions are untyped as well. Additionally the typing must commute in every typed graph morphism of a typed condition. Thus all its graphs must be typed by the same type graph. Any untyped condition can be transformed into a typed condition by typing all its graphs on $TG =$  .

Example 5. In Fig. 2.3 the satisfaction of $\exists($  $,\nexists($  $))$ on a typed graph G is shown. Importantly this condition holds on the given graph because, while there does exists an injective morphism r from the right side of the condition into G, it is not a typed graph morphism. To distinguish between the graphs of the untyped condition and their typing, the untyped nodes are given by numbers. Their typing is given explicitly by arrows into the type graph.

2.3. Graph Transformation

This section introduces graph transformation rules [EEPT06, HP09]. These are used to formalise change in graphs.

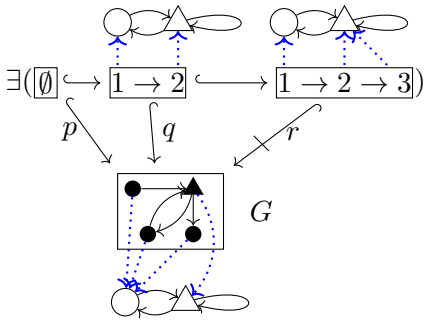


Figure 2.3.: Graph condition semantics on a graph

A transformation rule is characterized by three graphs. L is the left hand side to be matched. R is the right hand side resulting graph. K is the interface (or gluing) graph held in common between L and R.

Additionally, a transformation rule may contain restrictions to when it may be applied in the form of an application condition.

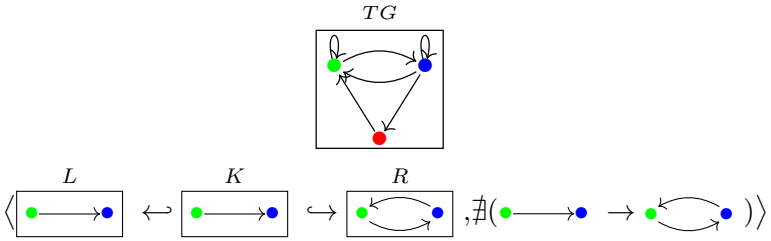
Definition 6 (transformation rule)**.** A **(typed) plain rule** $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ is described by three (typed) graphs connected via injective (typed) morphisms.

A **(typed) transformation rule** $\varrho = \langle p, ac \rangle$ consists of a plain rule p and a (typed) graph condition ac called the **application condition** of ϱ.

Given a plain rule p, a **direct derivation** of H from G by p is denoted by $G \Rightarrow_{p,m,m^*} H$ or $G \Rightarrow_p H$.

The **inverse rule** $p^{-1} = \langle R \hookleftarrow K \hookrightarrow L \rangle$ is created by swapping R and L.

Example 6. The following transformation rule takes a green node with an edge to a blue node and adds an inverted edge if it does not exist yet. The edges indicating the various typing morphisms are omitted because they are unambiguous.

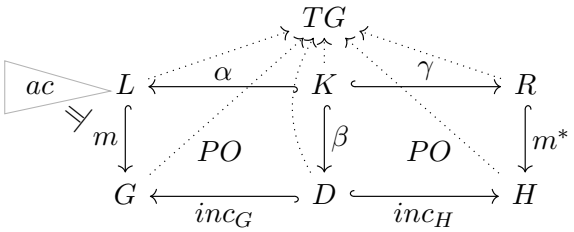


Definition 7 (rule application)**.** To apply a transformation rule ϱ to a graph G the following steps need to occur:

1. An injective morphism $m : L \hookrightarrow G$ that satisfies the application condition ac called a **match** is selected.

2. All elements matched by $L \setminus K$ are removed from G, yielding the pushout complement D with the inclusion morphism inc_G.

3. The pushout (H,m^*,inc_H) is constructed from γ and β yielding the output graph H with the comatch m^*.



Example 7. Fig. 2.4 shows the application of the previously defined transformation rule to an example graph.

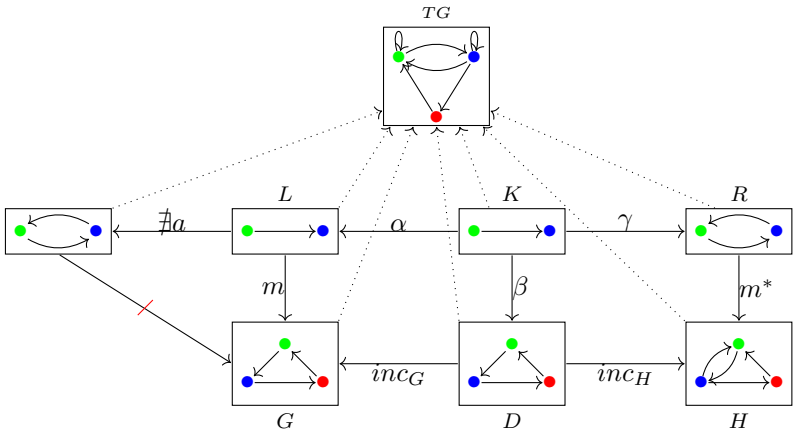


Figure 2.4.: Application of a rule to a graph.

Note. Upon deletion of elements from $L \setminus K$ edges may be incident to deleted nodes that are themselves not deleted. Because it is required for D to be a graph in this case the transformation can not be done. In [HS18] this is addressed by deleting all dangling edges using a graph program prior to rule application.

In order to repair a faulty model using these rules, it would be helpful to ensure a given rule will fix a violated constraint. A rule should also not violate new constraints in the process of fixing one. To achieve this, constraint preserving and constraint guaranteeing rules are introduced [HP09].

Definition 8 (guarantee and preservation)**.** Given a graph constraint c, a transformation rule ϱ is called c**-guaranteeing** if $\forall G \Rightarrow_\varrho H, H \models c$.

Given a graph constraint c, a transformation rule ϱ is called **c-preserving** if $\forall G \Rightarrow_\varrho H, G \models c$ implies $H \models c$.

Example 8. The rule `delete:` $\langle \bullet \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle$ is preserving for the constraint $c = \nexists(\bullet)$, because the set of graphs that satisfy c and can have the rule applied to them is empty. It is not guaranteeing, because a graph may have multiple nodes when the rule is applied.

Fact 2. [HP09] For all plain rules p and all constraints c there exist constructions Gua and $Pres$ such that $\langle p, Gua(p,c) \rangle$ and $\langle p, Pres(p,c) \rangle$ are c-guaranteeing and c-preserving, respectively.

2.4. Graph Programs

Graph programs use graph transformation rules to compute an output graph from a given input graph. Their are first introduced in [HP01] where a first definition is shown to be minimal and computationally complete. This definition is expanded in [PP13] to include if-then-else and try-then-else constructs. This framework uses graph programs for model repair, because they can directly incorporate the graph based constraints as part of individual rules that are combined to full repair programs. One specific programming language built on these principles is GP which is first introduced in [Ste07]. Its successor GP2 is shown to be compilable to efficient C-code in [BP16].

Definition 9 (graph program)**. Graph programs** are inductively defined:

1. Every finite set of (typed) transformation rules $\mathcal{R}$ is a program.
2. Given programs C, P and Q then
   a) $\langle P; Q \rangle$, (sequential composition)
   b) $P\downarrow$, (as-long as possible iteration)
   c) `if` C `then` P `else` Q, (branching statement)
   d) `try` C `then` P `else` Q, (branching statement)

   are programs.

The following rules and abbreviations are used:

$$
\begin{aligned}
&\texttt{skip} && \coloneqq \langle \emptyset \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle \\
&\varrho && \coloneqq \{\varrho\} \\
&\texttt{if } C \texttt{ then } P && \coloneqq \texttt{if } C \texttt{ then } P \texttt{ else skip} \\
&\texttt{try } C \texttt{ then } P && \coloneqq \texttt{try } C \texttt{ then } P \texttt{ else skip} \\
&\texttt{try } C && \coloneqq \texttt{try } C \texttt{ then skip}
\end{aligned}
$$

Definition 10 (graph program semantics)**.** A **configuration** is a tuple $\langle P,G\rangle$ of a program P and a graph G or a graph H where G signifies an unfinished computation and H signifies a finished computation.

A configuration γ is **terminal** if there is no configuration δ such that $\gamma \to \delta$.

A program P **finitely fails** on a graph G if there does not exist an infinite sequence $\langle P,G\rangle \to \langle P_1,G_1\rangle \to \dots$ and for each terminal configuration γ with $\langle P,G\rangle \Rightarrow^* \gamma, \gamma = \texttt{fail}$.

$\langle P,G\rangle \to^* H$ can also be written as $G \Rightarrow_P H$ or $\langle G,H\rangle \in [\![P]\!]$.

Two programs P,P' are **equivalent** ($P \equiv P'$) if for all transformations $G \Rightarrow_P H$ there exists a transformation $G \Rightarrow_{P'} H$ and vice versa.

A program P is **terminating** if the relation $\to$ is terminating.

The operational semantics of graph programs are given by the set of inference rules shown in Fig. 2.5.

$$[\texttt{Call}_1]\ \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R},G\rangle \to H} \qquad [\texttt{Call}_2]\ \frac{G \not\Rightarrow_{\mathcal{R}}}{\langle \mathcal{R},G\rangle \to \texttt{fail}}$$

$$[\texttt{Seq}_1]\ \frac{\langle P,G\rangle \to \langle P',H\rangle}{\langle\langle P;Q\rangle,G\rangle \to \langle\langle P';Q\rangle,H\rangle} \qquad [\texttt{Seq}_2]\ \frac{\langle P,G\rangle \to H}{\langle\langle P;Q\rangle,G\rangle \to \langle Q,H\rangle}$$

$$[\texttt{Seq}_3]\ \frac{\langle P,G\rangle \to \texttt{fail}}{\langle\langle P;Q\rangle,G\rangle \to \texttt{fail}}$$

$$[\texttt{If}_1]\ \frac{\langle C,G\rangle \to^+ H}{\langle \texttt{if } C \texttt{ then } P \texttt{ else } Q,G\rangle \to \langle P,G\rangle} \qquad [\texttt{If}_2]\ \frac{C \text{ finitely fails on } G}{\langle \texttt{if } C \texttt{ then } P \texttt{ else } Q,G\rangle \to \langle P,Q\rangle}$$

$$[\texttt{Try}_1]\ \frac{\langle C,G\rangle \to^+ H}{\langle \texttt{try } C \texttt{ then } P \texttt{ else } Q,G\rangle \to \langle P,H\rangle} \qquad [\texttt{Try}_2]\ \frac{C \text{ finitely fails on } G}{\langle \texttt{try } C \texttt{ then } P \texttt{ else } Q,G\rangle \to \langle Q,G\rangle}$$

$$[\texttt{Alap}_1]\ \frac{\langle P,G\rangle \to^+ H}{\langle P\downarrow,G\rangle \to \langle P\downarrow,H\rangle} \qquad [\texttt{Alap}_2]\ \frac{P \text{ finitely fails on } G}{\langle P\downarrow,G\rangle \to G}$$

Figure 2.5.: Inference rules for graph program commands

Example 9. Consider the rule `G_to_B:` $\langle \bullet \hookleftarrow \bullet \hookrightarrow \bullet \longrightarrow \bullet \rangle$ where the different coloured nodes distinguish different types. We construct two programs from this rule:

$$P_1 = \texttt{G\_to\_B}$$
$$P_2 = \texttt{G\_to\_B}\downarrow$$

For any input graph with a green node P_1 is terminating successfully. On inputs without a green node the program fails because the rule cannot be applied and no other rules can be chosen from.

P_2 terminates for any input graph without a green node because the iteration operator allows zero applications of the rule. On graphs with a green node the program is not terminating.

3. The Diagram Predicate Framework

This chapter introduces the Diagram Predicate Framework (DPF)[Rut10]. This modeling framework uses stacks of typed graphs to define the structure of a metamodel. Models are further refined by adding constraints as instantiations of predicate symbols.

3.1. Diagrammatic Meta-Modeling

DPF models are built on a stack of typed graphs having each layer typed by the previous one until the desired abstraction is reached on the lowest level.

Example 10. In Fig. 3.1 each meta model layer M_i can be expressed as a graph G_i being typed by the previous (more abstract) graph G_{i-1} indicated by a dotted line. The most abstract layer M_0 is typed by itself. The example describes an examination system in which a student may be enrolled in a course. An exam is defined by mapping the exam object to a student and a course.

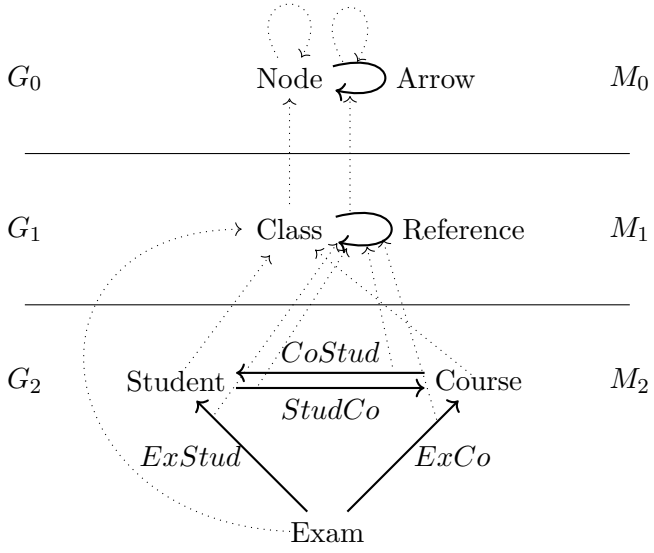


Figure 3.1.: Example metamodel stack

Stacked type graphs define the possible structure of model instances by defining their types and relations. Predicates are used to define further limitations on

the model. These act like a blueprint for specific graph constraints by defining their type and expected semantics.

Definition 11 (signature)**.** A **signature** $\Sigma = (P,\alpha)$ is a collection of **predicate symbols** P with a **map** α assigning a graph to each predicate symbol $p \in P$. An optional visualisation may be given.

The semantics of a signature Σ are given by a mapping that assigns each $p \in P$ a set $[\![p]\!]$ of graph morphisms $\iota : O \to \alpha(p)$ called **valid instances** of p, where O may vary over all graphs. $[\![p]\!]$ is assumed to be closed under isomorphisms.

Example 11. We want to incorporate the following requirements into our meta-model:

R1 A student who is appointed to an exam in a course must also be enrolled in that course.

R2 Whenever a student is enrolled in a course the course lists the student and vice versa.

In Table 3.1 the signature Σ_1 with two predicates [composite] and [inverse] is defined. These predicates can be instantiated into constraints for our model.

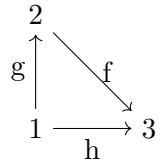
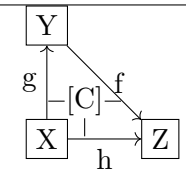
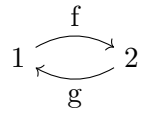
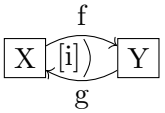
p	$\alpha^{\Sigma_1}(p)$	visualisation	Semantic Interpretation
[composite]			Whenever there exists an instance of h and an instance of g there must also exist an instance of f.
[inverse]			For each instance of f there exists an instance of g or vice versa.

Table 3.1.: Predicates of signature Σ_1

Note. The way of describing the set of valid instances is left entirely to the modeler. In our example we use set theoretic notation. The only restriction on the set is that it should be closed under isomorphism and that each graph in it requires a morphism into the predicates shape graph. Given that every graph has a morphism into a single node with a loop on it this does not limit the expressiveness at all.

Definition 12 (atomic constraint)**.** Given a signature $\Sigma = (P^\Sigma,\alpha^\Sigma)$, an **atomic constraint** (p,δ) added to a graph S is given by a predicate symbol p and a graph morphism $\delta : \alpha^\Sigma(p) \to S$.

Example 12. Table 3.2 shows how the informal requirements are formalised into constraints over the signature Σ_1. The mapping of each element in $\alpha^{\Sigma_1}(p)$ is given as a footnote in the corresponding element of $\delta(\alpha^{\Sigma_1}(p))$. Together with the model graph S these then yield a specification $\mathfrak{S}_2 = (S,C^{\mathfrak{S}_2} : \Sigma_1)$ as shown in Fig. 3.2.

Note that while the typing of Nodes is given by an arrow indicating the mapping of the typing morphism, the typing of edges is given as a label :< *type* >.

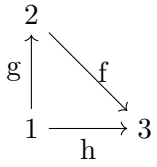
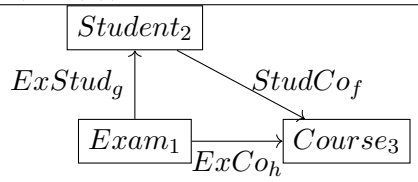
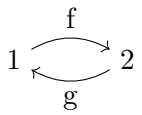
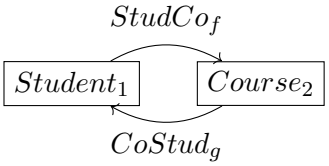
(p,δ)	$\alpha^{\Sigma_1}(p)$	$\delta(\alpha^{\Sigma_1}(p))$
([composite], δ_1)		
([inverse], δ_2)		

Table 3.2.: Atomic DPF constraints $C^{\mathfrak{S}_2}$ of $\mathfrak{S}_2$

Models are represented by specifications in DPF. They combine a graph together with atomic constraints. These constraints must be satisfied by instances of the specification.

Definition 13 (specification)**.** Given a signature $\Sigma = (P^\Sigma,\alpha^\Sigma)$ a **specification** $\mathfrak{S} = (S,C^\mathfrak{S} : \Sigma)$ is given by a graph S and a set $C^\mathfrak{S}$ of atomic constraints (p,δ) on S with $p \in P^\Sigma$.

Definition 14 (instance)**.** Given a specification $\mathfrak{S} = (S,C^\mathfrak{S} : \Sigma)$ an **instance** (I,ι) of $\mathfrak{S}$ is a graph I with a typing morphism $\iota : I \to S$ such that for each atomic constraint $(p,\delta) \in C^\mathfrak{S}$ there exists $\iota^* \in [\![p]\!]$ and $\iota^* : O^* \to \alpha^\Sigma(p)$ is given by a pullback:

$$\begin{array}{ccc} \alpha^\Sigma(p) & \xrightarrow{\delta} & S \\ {\scriptstyle \iota^*}\uparrow & PB & \uparrow{\scriptstyle \iota} \\ O^* & \xrightarrow[\delta^*]{} & I \end{array}$$

A graph typed by S but not satisfying all model constraints is called a **candidate instance** of $\mathfrak{S}$.

Example 13. Fig. 3.2 shows the specification $\mathfrak{S}_2$ incorporating the constraints as shown by their visualisation. A candidate instance of the model I is given with its typing morphism. The types of edges are given as :< *type* > here to not clutter the image with typing arrows.

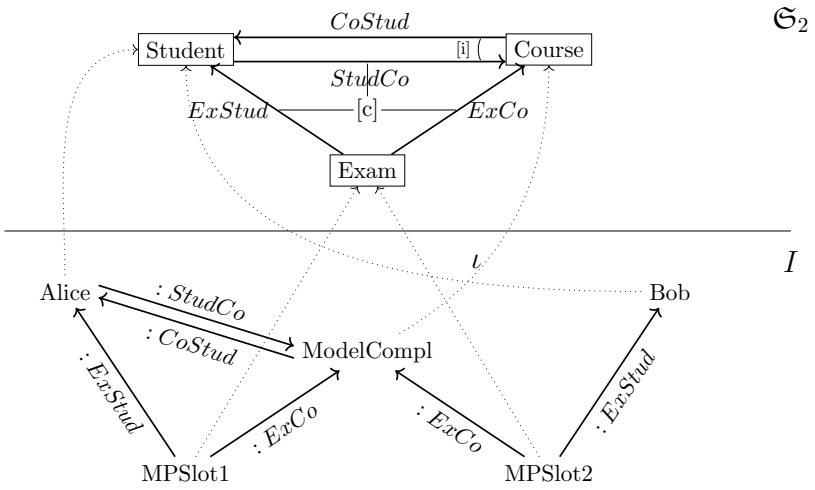


Figure 3.2.: Specification $\mathfrak{S}_2 = (S, C^{\mathfrak{S}_2} : \Sigma_1)$ and a candidate instance I

3.2. Finding Inconsistencies

Rabbi et al. extend DPF with a method for finding and repairing errors in a candidate instance of a model. To achieve this they use predefined graph transformation rules for possible identified violations [RLYK15].

1. Take some partial model I typed by a metamodel $\mathfrak{S}_i$ and a constraint $\alpha^{\Sigma_{i-1}}$ created from the predicate $[pred]$ of Σ_{i-1}.
2. Extract the pullback from $\alpha^{\Sigma_{i-1}}([pred]) \xrightarrow{\delta_i} S \xleftarrow{\iota} I$
3. This yields a graph O^* with morphisms $O^* \xrightarrow{\iota^*} \alpha^{\Sigma_{i-1}}([pred])$ and $O^* \xrightarrow{\delta_1^*} I$
4. The morphism ι^* must be in the set of valid instances $\alpha^{\Sigma_{i-1}}([pred])$. Otherwise there exists an error in this candidate instance of the model.

The pullback on the constraints morphism δ_i and the typing morphism ι extracts the part of the instance graph that is relevant to the constraint. This instance graph and its morphism ι^* to the constraint hold the information if the semantics of the constraint are satisfied.

Example 14. In the case of our example doing this for the ([composite],δ_1) constraint yields the output shown in Fig. 3.3. Here ι^* does not satisfy the semantics of the constraint as the graph is missing an edge from Bob to the course. The subgraphs of the model and its instance that are relevant to the constraint are marked red.

Rutle et al. categorize constraints by their origin and their effect in [RRLW12]. Constraints may originate in the modeling language (structural constraints) or be

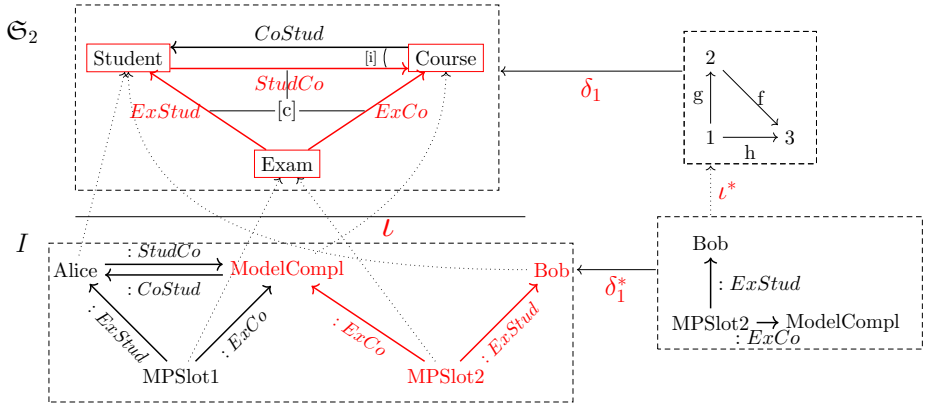


Figure 3.3.: Example output of constraint pullback.

added by constraint languages (attached constraints). By this definition the typing of a model is in itself a structural constraint. When distinguishing by effect, constraints may be satisfied by the models themselves and those that should be satisfied by instances of a model.

DPF aims to unify these constraints as part of its modeling language as the predicate-constraint approach introduced above. Additionally they define **universal constraints** which are constraints that every definable specification in a modeling formalism must conform to.

One issue with this approach and the reason for the adaptations made in this thesis is that no restrictions on the semantics definitions for a predicate exist. As long as the set of valid instances is fully described, the methods used for their definition is free. When trying to automatically check for constraint satisfaction, problems may arise with definitions differing in form and complexity.

4. A Condition Based Modeling Framework

This chapter introduces a Modeling and Repair Framework (MR) based on the Diagram Predicate Framework [Rut10]. The main goal is to provide uniform formal semantics for all model constraints. Satisfaction of a model should be decidable in general and models should be repairable. To achieve this we replace predicates with graph conditions that may describe constraints on any model layer.

4.1. Running Example

The Modeling and Repair Framework (MR) is introduced along an example to illustrate various points. The task is to model the organization of a company workforce given by the requirements below.

- **R1** Every person working in the company is an employee.
- **R2** Every employee has a unique personal number (ID).
- **R3** Every employee has one or more positions in the company.
- **R4** Every employee is part of exactly one team.
- **R5** Every team has exactly one team leader.
- **R6** A team may only be led by someone with the team leader position.
- **R7** Teams are distinguished into administrative and executive teams.
- **R8** Administrative teams may include the positions team leader, temporary employee and staff.
- **R9** Executive teams may include the positions team leader, temporary employee, senior developer and junior developer.
- **R10** Temporary employees may not be team leaders.

The derived meta-model including the constraint visualisations is shown in Fig. 4.1. It consist of four model layers where each layer is typed by the previous one. To avoid an excess amount of typing arrows from S_3 to S_2 the typing is given instead by colour coding vertices and their corresponding type vertex equally. Edge types in S_3 are provided as subscripts on their name. **R1**, **R7**, **R8** and **R9** are enforced directly by the structure of the type graphs.

Additionally the signs u,1! and ≥ 1 in S_2 are the visualisations of multiplicity constraints in the model. They enforce the requirements **R2**, **R3**, **R4** and **R5**. The constraints indicated by $\overline{\wedge}$ and ! in S_3 enforce the requirements **R6** and **R10**.

The precise definition and semantics of these constraints is explained in the remainder of this chapter.

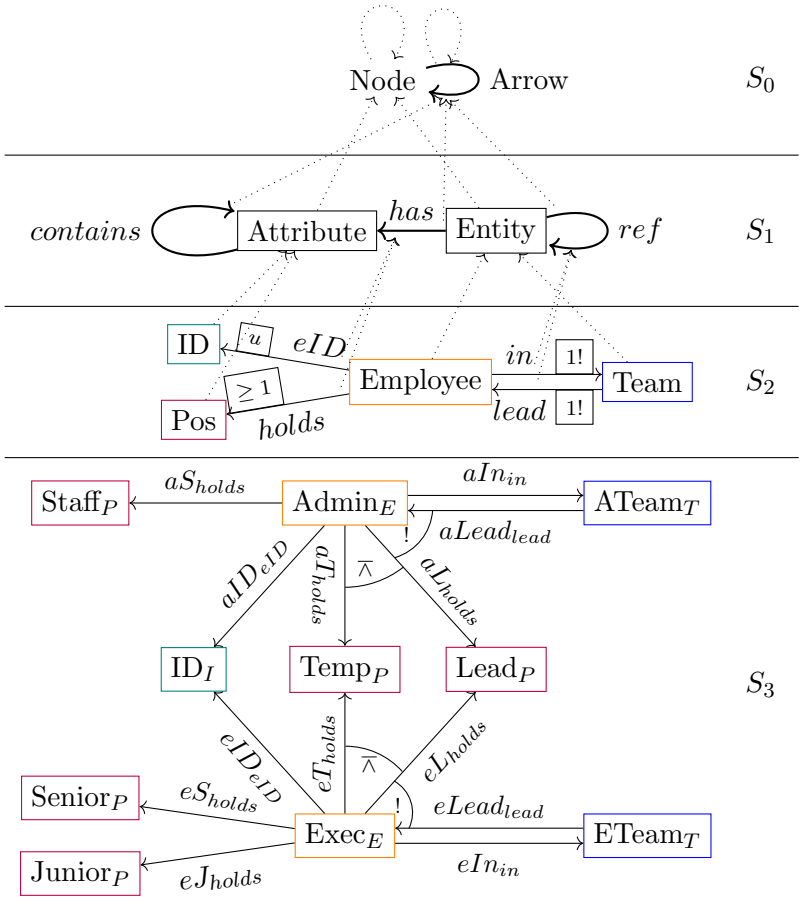


Figure 4.1.: Metamodel stack for company example.

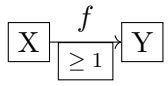
4.2. MR Modeling

In Chapter 3 different constraint types and their use in DPF are discussed. It is established that the freedom of defining predicate semantics allows for very complex constraints. This becomes an issue however when trying to automatically check if a constraint is satisfied and also generating programs that repair it. For this reason the predicate definition is altered to be limited to nested graph conditions.

Our approach removes the need for shape graphs and explicit semantics interpretations as both the shape and semantics are contained within the condition itself. Since these graph conditions are blueprints for constraints on the model they are referred to as templates.

Definition 15 (template)**.** A **template** $\tau = (c,t)$ consists of an untyped graph condition c and a name t. A **signature** Σ is a set of templates. A template may optionally be provided with a visual representation for easier identification in a specification.

Example 15. The following template informally requires the existence of at least one outgoing edge f on any node 1. In a given model it may be visualised by adding "≥ 1" to an edge it should apply to. 1 and 2 refer to any untyped node. We choose numbers for nodes and letters for edges to easier identify them when they are typed.

name t	condition c	visualisation
[atLeastOne]	$\forall(1,\exists(1 \xrightarrow{f} 2))$	

Definition 16 (constraints)**.** Given a signature Σ and a graph S, an **atomic (model) constraint** (τ,J) is a template $\tau \in \Sigma$ together with a set of typing morphisms J. This set is inductively defined from the templates graph condition c via the operator $cons(c,S)$ as follows:

- $cons(true,S) = \emptyset$
- $cons(c_1 \wedge c_2,S) = cons(c_1,S) \bigcup cons(c_2,S)$
- $cons(\neg c,S) = cons(c,S)$
- $cons(\exists(a,c),S) = \{\iota_1 : P \to S,\ \iota_2 : C \to S\} \bigcup cons(c)$ with $a : P \hookrightarrow C$ such that $a^* : (P,\iota_1) \to (C,\iota_2)$ constitutes a typed graph morphism.

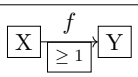
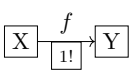
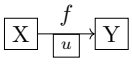
A **specification** $\mathfrak{S} = (S,X^{\mathfrak{S}} : \Sigma)$ is given by a typed graph S and a set $X^{\mathfrak{S}}$ of atomic constraints on S derived from the same signature Σ.

Observation 1. Graph condition based templates and constraints allow us to describe first-order properties [HP09]. Binding a condition to a higher model level does not yield more expressive constraints as any higher level constraint may be replaced with a disjunction of lower level constraints Algorithm 1.

An advantage of allowing constraints on every level is a more concise definition of overarching properties.

Example 16. Table 4.1 shows the necessary predicates and constraints used to satisfy the example model requirements. Note that the requirement for every employee to have a position as well as the uniqueness of the team relation are defined a level above the other predicates and are part of a different signature. This allows for positions and teams to be created on the underlying layer without changes to overarching constraints.

The nodes of each template graph are given by numbers and their edges by single letters. These are added as subscripts in their corresponding types in the constraint graphs.

name t	condition c	visualisation	derived constraints in $\mathfrak{S}_2$
[atLeastOne]	$\forall(1,\exists(1 \xrightarrow{f} 2\ ))$		$\forall(\text{Employee}_1,\exists(\ \text{Employee}_1 \xrightarrow{holds_f} \text{Pos}_2\ ))$
[exactlyOne]	$\forall(1,\exists(1 \xrightarrow{f} 2))$ $\wedge \nexists(1 \underset{g}{\overset{f}{\rightrightarrows}} 2\ 3))$ $\wedge \nexists(1 \underset{g}{\overset{f}{\rightrightarrows}} 2))$		$\forall(\text{Employee}_1,\exists(\text{Employee}_1 \xrightarrow{in_f} \text{Team}_2\ ))$ $\wedge \nexists(\text{Employee}_1 \overset{in_f}{\underset{in_g}{\rightrightarrows}} \text{Team}_2\ \text{Team}_3)$ $\wedge \nexists(\text{Employee}_1 \underset{in_g}{\overset{in_f}{\rightrightarrows}} \text{Team}_2\ )$
			$\forall(\text{Team}_1,\exists(\text{Team}_1 \xrightarrow{lead_f} \text{Employee}_2))$ $\wedge \nexists(\text{Team}_1 \overset{lead_f}{\underset{lead_g}{\rightrightarrows}} \text{Employee}_2\ \text{Employee}_3)$ $\wedge \nexists(\text{Team}_1 \underset{lead_g}{\overset{lead_f}{\rightrightarrows}} \text{Employee}_2\ )$
[unique]	$\forall(1,\exists(1 \xrightarrow{f} 2))$ $\wedge \nexists(1 \underset{g}{\overset{f}{\rightrightarrows}} 2\ 3)))$ $\wedge \nexists(1 \underset{g}{\overset{f}{\rightrightarrows}} 2))$ $\wedge \nexists(\begin{smallmatrix}1\\2\end{smallmatrix} \underset{g}{\overset{f}{\rightrightarrows}} 3))$		$\forall(\text{Employee}_1,\exists(\text{Employee}_1 \xrightarrow{eID_f} \text{ID}_2))$ $\wedge \nexists(\text{Employee}_1 \overset{eID_f}{\underset{eID_g}{\rightrightarrows}} \text{ID}_2\ \text{ID}_3)$ $\wedge \nexists(\text{Employee}_1 \underset{eID_g}{\overset{eID_f}{\rightrightarrows}} \text{ID}_2\ )$ $\wedge \nexists(\begin{smallmatrix}\text{Employee}_1\\ \text{Employee}_2\end{smallmatrix} \underset{eID_g}{\overset{eID_f}{\rightrightarrows}} \text{ID}_3\ )$

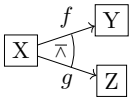
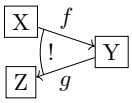
name t	condition c	visualisation	derived constraints in $\mathfrak{S}_3$
[notTwo]	$\nexists(1 \underset{g}{\overset{f}{\rightrightarrows}} 2\ 3))$		$\nexists(\ Admin_1 \overset{aT_f}{\underset{aL_g}{\rightrightarrows}} Temp_2\ Lead_3)$
			$\nexists(\ Exec_1 \overset{eT_f}{\underset{eL_g}{\rightrightarrows}} Temp_2\ Lead_3)$
[mustFollow]	$\forall(\ 1 \xrightarrow{f} 2\,,$ $\exists(\begin{smallmatrix}1\\3\end{smallmatrix} \underset{g}{\overset{f}{\rightrightarrows}} 2))$		$\forall(\text{Admin}_1 \xrightarrow{aLead_f} \text{ATeam}_2$ $\exists(\ \text{ATeam}_1 \overset{aLead_f}{\underset{\text{aL}_g}{\rightrightarrows}} \text{Admin}_2\ \text{Lead}_3))$
			$\forall(\text{Exec}_1 \xrightarrow{eLead_f} \text{ETeam}_2$ $\exists(\ \text{ETeam}_1 \overset{eLead_f}{\underset{eL_g}{\rightrightarrows}} \text{Exec}_2\ \text{Lead}_3))$

Table 4.1.: Signatures Σ_2 (top three templates) and Σ_3(bottom 2 templates) with derived constraints resulting in $\mathfrak{S}_2$ and $\mathfrak{S}_3$

Definition 17 (model)**.** A **model** $\mathcal{M} = (\mathcal{S},Sig,Spec)$ consists of a list of typed graphs $\mathcal{S} = [S_0,S_1,\ldots,S_n],n > 0$ where $\forall i \in [1,\ldots,n]S_i$ is typed by S_{i-1} and S_0 is typed by itself, a list of signatures $Sig = [\Sigma_0,\ldots,\Sigma_n]$ with one (possibly empty) signature used on each layer and a list $Spec = [\mathfrak{S}_0,\ldots,\mathfrak{S}_n]$ of corresponding constraint specifications.

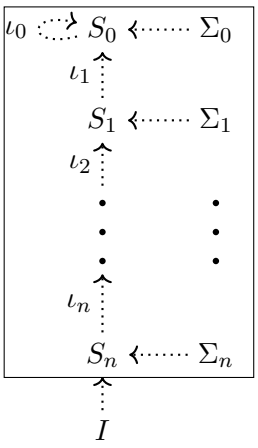


Figure 4.2.: Abstract model with n layers and instance I

Example 17. Fig. 4.1 shows a full model with four layers S_0 to S_3. It includes two non-empty signatures Σ_2,Σ_3 and their corresponding specifications $\mathfrak{S}_2$,$\mathfrak{S}_3$ by their visualisations defined in Table 4.1

Observation 2. Any single typed graph condition can become a model by using the identity morphism as the typing of its type graph.

Definition 18 (instance)**.** Given a Model $\mathcal{M} = (\mathcal{S},Sig,Spec)$, a **candidate instance** $I = (IG,\mathcal{M})$ is a typed graph $IG = (G,\iota_G)$ with $\iota_G : G \to S_n$ where S_n is the last graph in $\mathcal{S}$.

I is a **valid instance** $I \in [\![\mathcal{M}]\!]$ if (G,ι_G) satisfies the constraints in $\mathfrak{S}_n$ and $\forall i \in [1,n](G,\iota_i \circ \iota_{i+1} \circ \cdots \circ \iota_n \circ \iota_G)$ satisfies the constraints in $\mathfrak{S}_{i-1}$.

Two models $\mathcal{M}_1$ and $\mathcal{M}_2$ are **equivalent** $\mathcal{M}_1 \equiv \mathcal{M}_2$ if $\mathcal{S}_1 = \mathcal{S}_2$ and for all typed graphs IG $(IG,\mathcal{M}_1) \in [\![\mathcal{M}_1]\!] \implies (IG,\mathcal{M}_2) \in [\![\mathcal{M}_2]\!]$ and vice versa.

Example 18. Fig. 4.3 shows a candidate instance of the example model. It is typed correctly by S_3 but violates the position constraint that an employee may not be both temporary and a team leader. Typing arrows of employee, ID and team nodes have been omitted for visual clarity.

Fact 3 (Model checkability)**.** Since each constraint can be separately checked for satisfaction, satisfaction of a model can be checked for any given candidate instance.

Theorem 1 (Undecidability of model equivalence)**.** Given two models $\mathcal{M}_1$ and $\mathcal{M}_2$ it is undecidable in general whether $\mathcal{M}_1 \equiv \mathcal{M}_2$.

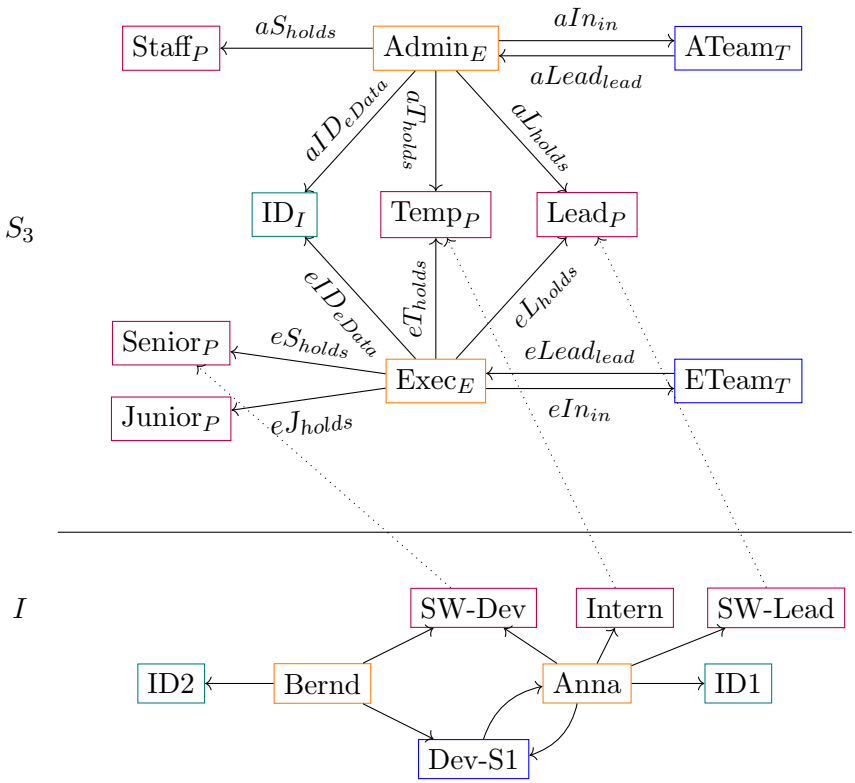


Figure 4.3.: Candidate instance of the running example model

This can be shown by reducing the problem to decidability of graph condition equivalence.

Proof (Theorem 1)**.** Let the models $\mathcal{M}_1$ and $\mathcal{M}_2$ be defined on the single layer type graph stack:

Any constraint can be instantiated on S_0. Let $\mathcal{M}_1$ contain an arbitrary constraint c_1 and $\mathcal{M}_2$ an arbitrary constraint c_2. By Definition 18 $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent if they are defined on the same typed graph stack and all typed graphs that are instances of $\mathcal{M}_1$ are instances of $\mathcal{M}_2$ and vice versa.

The first point holds by definition.

Any graph can be instantiated on S_0. Thus the models are equivalent if and only if any graph satisfying c_1 also satisfies c_2 and vice versa. That is if $c_1 \equiv c_2$. The equivalence problem for graph conditions is undecidable [Pen09]. Thus model equivalence is undecidable in general. □

4.3. Case Studies

In this section we present examples of meta models for established graph based system representations. This is to establish whether they can be emulated in MR and which extensions could be made to make it possible if they can not.

Hierarchical Finite State Machines

Hierarchical finite state machines are a common tool for modeling system behaviour in software engineering. This is an attempt to encode one version of such a machine into MR. Fig. 4.4 shows an example machine together with the derived meta-model stack and the corresponding instance as a typed graph.

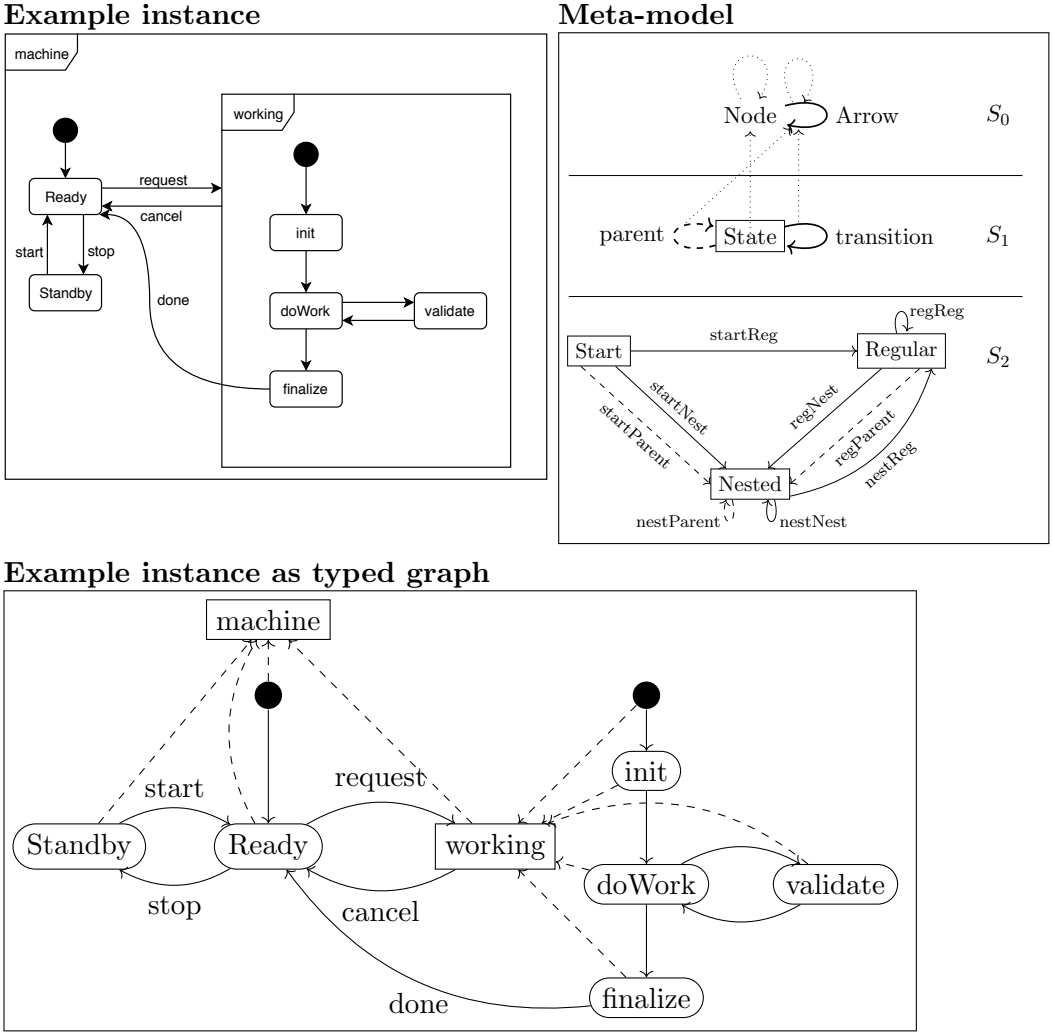


Figure 4.4.: Example hierarchical state machine with meta-model stack

Here a state may contain another nested state machine that is active as long as the state is active in the parent automaton. For modeling the structure, three types of states are established. **Regular** states are non-nested states with any incoming or outgoing state transitions. **Start** states point to the first state of a nested state and may not have incoming edges. **Nested** states contain a start state and any number of regular states. The boxes in Fig. 4.4 signify which nested state any given sub-state belongs to. In the model, this is represented using dashed edges connecting each state to its parent. Additionally the following requirements should hold:

R1 There is only one state that is its own parent and every state has a parent.

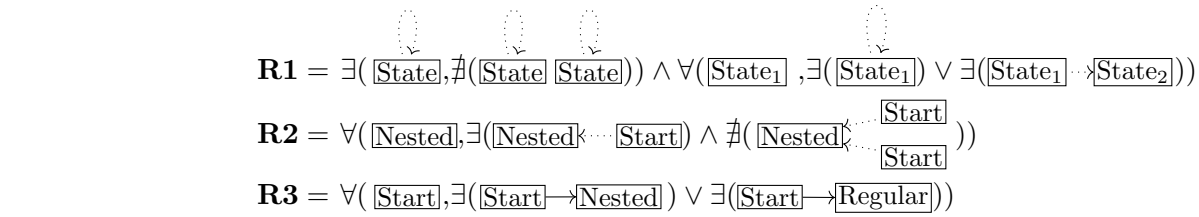
R2 Every nested state has exactly one start state.

R3 A start state has an outgoing edge to a sibling node.

R4 Following parent relations always leads to the top nested state.

R5 Parent relations are loop free except for the trivial loop on the top.

With graph conditions it is possible to encode **R1**, **R2** and **R3**. **R4** and **R5** can not be expressed as they are not first-order properties.



Control/Data Flow Graphs

Control and Data Flow Graphs (CDFG) can model both operational control and data dependencies for a system. Fig. 4.5 shows an example graph as well as our constructed meta-model. The control flow determines the execution order of basic blocks (BB). It may contain branching statements (`IF`) and loops, modeling the operational system part.

Basic blocks contain operations that may be executed in arbitrary order, but only if their data dependencies (or values) are satisfied. For example **Op3** in **BB3** may only be executed once **Op1** and **Op2** of the same block are finished. Operations must form directed acyclic graphs.

Similar to hierarchical state machines we use a dashed edge to link operations to a common basic block. Additionally we add start and end nodes to basic blocks (green in the meta-model). With this we establish the following basic requirements for the model:

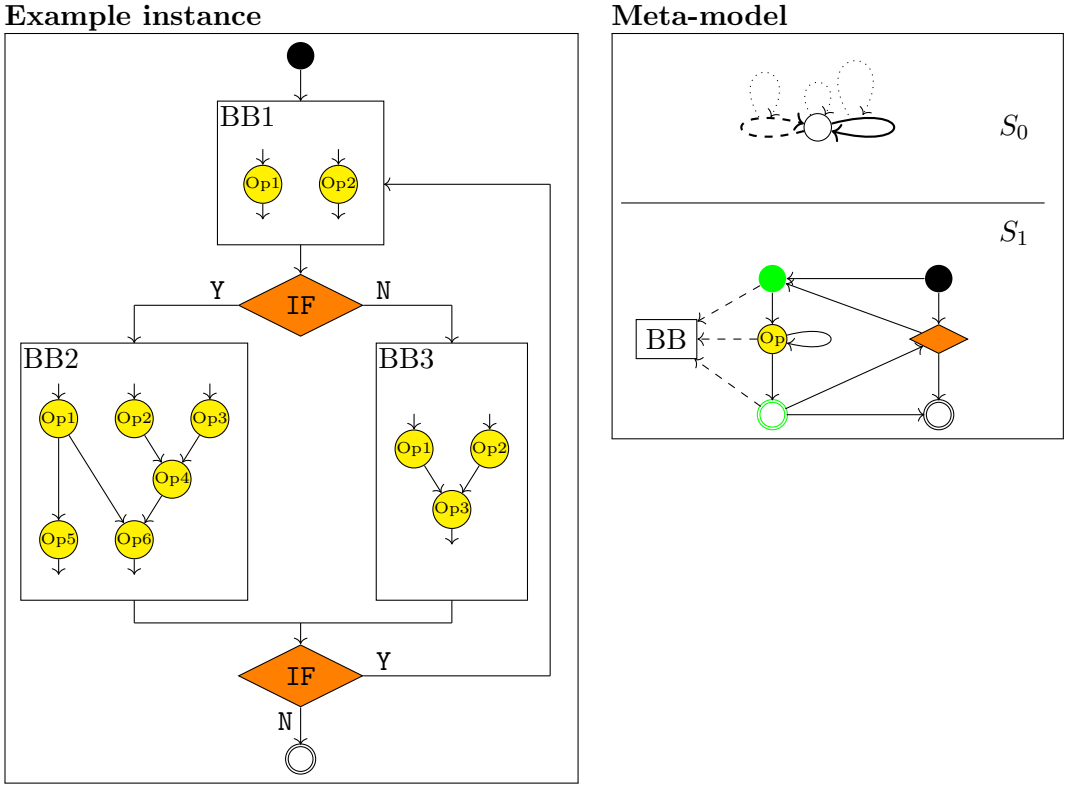


Figure 4.5.: Example Control and Data Flow Graph

R1 Bounding boxes have unique start and end nodes.

R2 A CDFG has a unique start and end node.

R3 The end node is reachable from any node.

R4 Operations within a bounding box are acyclic.

R1 and **R2** can be expressed with graph conditions similar to the hierarchical state machine example. **R3** and **R4** can not be expressed because they are not first-order properties.

Note. R3 is also special in that it would quantify over all nodes but then has to reference the end node. Even if path properties were possible in graph conditions this would require a conjunction of constraints for every sub-type of node currently. It may be desirable to have a feature that allows "jumping" levels in a constraints where adequate. For **R3** this would mean:

$$\mathbf{R3} = \forall(\bigcirc,\exists(\bigcirc \xrightarrow{path} \bigcirc\!\!\bigcirc))$$

4.4. Common Constraint Types

Through our case studies we have established that MR can model complex systems structurally. The limitation to first-order constraints, especially the lack of path and loop constraints is however very relevant.

To identify how expressive a modeling framework needs to be for general use, we determine a set of commonly used constraints in software engineering in Table 4.2. This list is based on works on modeling [Rut10, RLYK15] and the authors own experience. The information if a constraint is expressible in graph condition (GC) or monadic second order conditions (MSO) is provided based on [Cou97, CE12]. The final examples in the table are written in braces, because they are not too common but show the limitations of what is expressible in monadic-second-order. This is particularly relevant as by Courcelle's theorem any property definable in monadic second-order logic is decidable in linear time on graphs of bounded tree width [Cou90]. As constraints may get arbitrarily complex depending on the use case this list is by no means exhaustive.

Name	Description	GC	MSO
structure exists	There exists a given finite structure within the graph.	✓	✓
structure does not exist	There exists no occurrence of a given finite structure within the graph.	✓	✓
multiplicity(n,m)	There exist between n and m instances of an edge type X on any node of type Y.	✓	✓
irreflexive	An edge type may not have the same source and target.	✓	✓
inverse	For every edge 1 $\xrightarrow{f}$ 2 there exists an inverse edge 1 $\xleftarrow{g}$ 2.	✓	✓
disjoint image	The set of targets of edge types X and Y are disjoint.	✓	✓
image inclusion	Every node that is a target of an edge of type X is also target of an edge of type Y.	✓	✓
image equal	The set of targets of edges typed X and Y are equal.	✓	✓
injective	Edges of type X never share a target.	✓	✓
surjective	Every node of type Y is the target of an edge of type X.	✓	✓
jointly injective	Edges of edge type Y or Z, originating from a node of type X, never share a target.	✓	✓
jointly surjective	Every node of type Z is the target of an edge of type X or Y.	✓	✓
total	Every node of type X has an outgoing edge of type Y.	✓	✓
bijective	injective, surjective and total	✓	✓
commutative	Whenever there exist edges f and g of a given type from X over Y to Z there also exist edges f' and g' from X over Y to Z.	✓	✓
path	There exists a path from node X to node Y.	✗	✓
loop	The edge type f forms a loop on X.	✗	✓
set comparison	The number of nodes of type X and Y is equal.	✗	✗
(set counting)	The number of nodes of type X is even/odd/prime.	✗	✗
(hamiltonian)	The graph has a hamiltonian cycle.	✗	✗

Table 4.2.: Common constraints in software engineering

5. Repair

This chapter contains the procedure for repairing candidate instances of MR models. This is done by constructing a repair program that will transform any candidate instance of a model into a valid instance.

Definition 19 (repair program)**.** A graph program P is a **constraint repair program** for a graph constraint c if for all transformations $G \Rightarrow_P H$,$H \models c$.
It is a **model repair program** for a model $\mathcal{M}$ if for all candidate instances I_c: $I_c \Rightarrow_P I_v$ outputs a valid instance $I_v \in [\![\mathcal{M}]\!]$.

5.1. Propagating Constraints

Since MR models are entirely graph based both approaches to graph repair and model repair are considered. Due to the nature of these models three main challenges present themselves for the repair:

1. A constraint on meta-level i may effectively result in many constraints on level $i+1$. One constraint for each possible combination of the sub-types of elements used in S_i as described in Algorithm 1.

2. Repairing a constraint only on its respective level may leave newly created elements untyped on lower levels.

3. The modeler may want to differentiate the repair rules for a constraint on meta-level i between its derived sub-types on level $i+1$.

Any constraint on a higher than bottom level can effectively be replaced by one on a lower model level without changing the model semantics. This can be used to replace any model by an equivalent one where all constraints are on the bottom level.

Theorem 2 (Equivalence preserving replacement of constraints on a lower level)**.** Given a model $\mathcal{M} = (\mathcal{S},Sig,Spec)$ with type graphs $\mathcal{S} = [(S_0,\iota_{S_0}),\ldots,(S_n,\iota_{S_n})]$, signatures $[\Sigma_0,\ldots,\Sigma_n]$ and specifications $[\mathfrak{S}_0,\ldots,\mathfrak{S}_n]$, a model $\mathcal{M}^* = cr(\mathcal{M},c)$ with constraint $c = (\tau,J_c)$, $c \in \mathfrak{S}_i$, $i < n$ can be constructed such that:

- $\mathcal{M} \equiv \mathcal{M}^*$ and $\mathcal{S} = \mathcal{S}^*$ and $\mathfrak{S}_j = \mathfrak{S}_j^*$ for all $j < i$ or $j > i+1$
- $c \notin \mathfrak{S}_i^*$

Note. The satisfaction of a constraint in a given instance can be checked by interpreting its elements as elements of the corresponding model level. This means that to pull the constraint to a lower level it has to hold on all its possible derived types. Thus, the main idea for constructing this model is to replace the constraint by a disjunction of constraints on the same template. Each element of the disjunction being a possible sub-typing of original constraint.

If no lower level typing can be found for a given morphism the set is empty and the corresponding sub-constraint is $false$ as no model may satisfy a positive constraint it can not be typed on. Since the resulting constraint generally varies in shape from the original a new template derived as the base of the constructed constraint must be added to the lower level specification.

Example 19. Consider the example in Fig. 5.1. It shows a constraint for the existence of an edge on S_0 pulled down to S_1. This is done by creating a disjunction of the original template for every possible sub-typing of the targets of ι_a in S_1. This yields the typing morphisms ι_d^A and ι_d^B of a new template, creating a new constraint that is equivalent on the given meta-model.

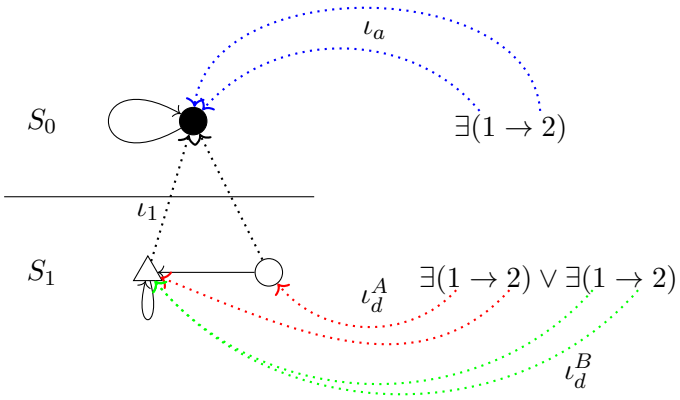
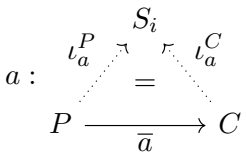


Figure 5.1.: Example for pulldown construction idea

Construction 1 (Constraint Replacement)**.** First we construct a new constraint $pd(\mathcal{M},c) \in \mathfrak{S}_{i+1}$ inductively on the definition of typed graph conditions.

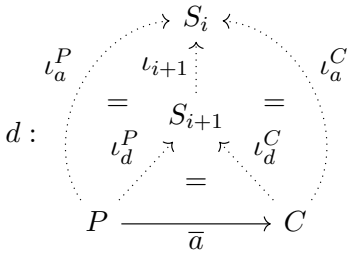
- $pd(\mathcal{M},true) = true$
- $pd(\mathcal{M},c_1 \wedge c_2) = pd(\mathcal{M},c_1) \wedge pd(\mathcal{M},c_2)$
- $pd(\mathcal{M},\neg c) = \neg pd(\mathcal{M},c)$
- $pd(\mathcal{M},\exists(a,c)) = \begin{cases} \bigvee_{d \in \mathcal{D}} \exists(d,pd(\mathcal{M},c)) & \text{if } \mathcal{D} \neq \emptyset \\ false & \text{otherwise} \end{cases}$

Here each $d \in \mathcal{D}$ is derived from a typed graph morphism $a : (P,\iota_a^P) \to (C,\iota_a^C)$ in c where $\tau = (base,t)$ is the template c is defined on and P,C are graphs in $base$ with corresponding typing morphisms ι_a^P,ι_a^C from constraint c.



$\mathcal{D}$ is the maximal set of typed graph morphisms such that:

- $\forall d \in \mathcal{D}$
  - $d : (P,\iota_d^P) \to (C,\iota_d^C)$, $\iota_d^P : P \to S_{i+1}$, $\iota_d^C : C \to S_{i+1}$ and
  - $\iota_{i+1} \circ \iota_d^P = \iota_a^P$ and
  - $\iota_{i+1} \circ \iota_d^C = \iota_a^C$
- $\forall d_1,d_2 \in \mathcal{D}\ d_1 \neq d_2 \implies \exists e \in C_0 \bigcup C_1 : \iota_{d_1}^C(e) \neq \iota_{d_2}^C(e)$

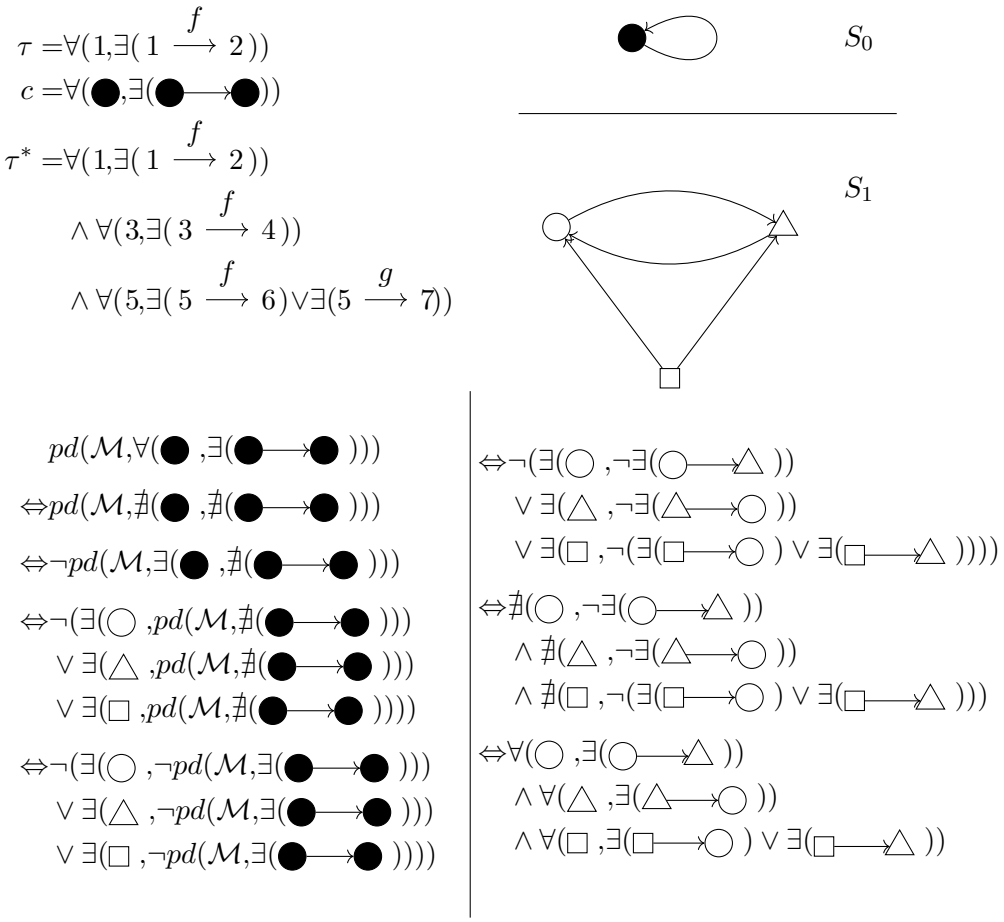


Finally let $\tau^* = (base^*,t^*)$ be the resulting template where $base^*$ is the resulting untyped base condition of $pd(\mathcal{M},c)$ and $t^* =$ 'pd-' $+ t$.

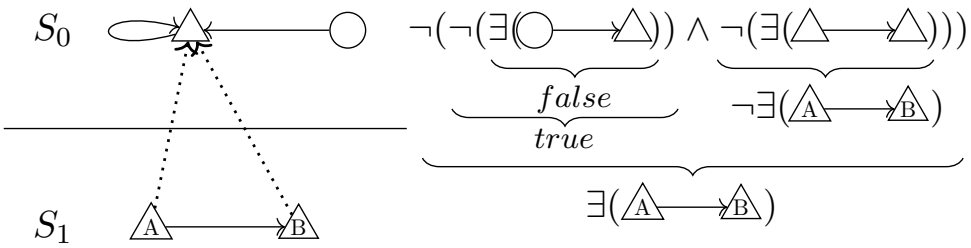
With this $cr(\mathcal{M},c) = \mathcal{M}^* = (\mathcal{S}^*,Sig^*,Spec^*)$ such that:

- $\mathcal{S}^* = \mathcal{S}$
- $Sig_o^* = Sig_o$ for $o \neq i+1$
- $Sig_{i+1}^* = Sig_{i+1} \cup \tau^*$
- $\mathfrak{S}_o = \mathfrak{S}_o^*$ for $o < i$ or $o > i+1$
- $\mathfrak{S}_i^* = \mathfrak{S}_i \setminus c$
- $\mathfrak{S}_{i+1}^* = \mathfrak{S}_{i+1} \bigcup pd(\mathcal{M},c)$

Example 20. Fig. 5.2a shows a small meta-model. We add a single constraint on S_0 that each node must have an outgoing edge. This constraint can be pulled down to S_1 in the steps shown. The resulting constraint gives a conjunction for each sub-type of a node to have an outgoing edge. Fig. 5.2b shows a second construction where one sub-constraint becomes $false$ resulting in a smaller constraint.



a) Meta-model and pulldown construction step by step



b) Pulldown construction with $false$.

Figure 5.2.: Pulldown construction examples

Proof (Theorem 2)**.** By Definition 18, two models $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent if:

1. Their type graph stacks are the same $\mathcal{S}_1 = \mathcal{S}_2$ and
2. for all typed graphs IG $(IG,\mathcal{M}_1) \in [\![\mathcal{M}_1]\!] \implies (IG,\mathcal{M}_2) \in [\![\mathcal{M}_2]\!]$ and vice versa.

Point 1 holds by definition in Construction 1 as $\mathcal{S}^* = \mathcal{S}$.
For point 2 it suffices, without loss of generality, to show that an instance I that satisfies c must satisfy $pd(\mathcal{M},c)$ and vice versa, because no other constraints differ.

Let (Q,ι_Q) be a typed graph with $\iota_Q : Q \to S_n$ and ϕ_i the direct typing morphism from Q to meta-graph S_i.

$$\phi_i = \iota_{i+1} \circ \iota_{i+2} \circ \cdots \circ \iota_n \circ \iota_Q \qquad (DT)$$

$$\text{to be shown: } (Q,\phi_i) \models c \Leftrightarrow (Q,\phi_{i+1}) \models pd(\mathcal{M},c)$$

case 1: $c = true$
For $c = true$ the typing always commutes as the base graph is empty. The condition holds for every typed graph morphism (Definition 5). Thus:

$$(Q,\phi_i) \models true \Leftrightarrow (Q,\phi_{i+1}) \models pd(\mathcal{M},true) = true$$

case 2: $c = \neg c_1$
By induction hypothesis, the claim holds for c_1 (IH).

$$\begin{aligned}
&(Q,\phi_i) \models \neg c_1 \\
\Leftrightarrow &(Q,\phi_i) \not\models c_1 && (Definition\ 5)\\
\Leftrightarrow &(Q,\phi_{i+1}) \not\models pd(\mathcal{M},c_1) && (IH)\\
\Leftrightarrow &(Q,\phi_{i+1}) \models \neg pd(\mathcal{M},c_1) && (Definition\ 5)\\
\Leftrightarrow &(Q,\phi_{i+1}) \models pd(\mathcal{M},\neg c_1) && (Construction\ 1)
\end{aligned}$$

case 3: $c = c_1 \wedge c_2$
By induction hypothesis, the claim holds for c_1 and c_2 (IH).

$$\begin{aligned}
&(Q,\phi_i) \models c_1 \wedge c_2 \\
\Leftrightarrow &(Q,\phi_i) \models c_1 \wedge (Q,\phi_i) \models c_2 && (Definition\ 5)\\
\Leftrightarrow &(Q,\phi_{i+1}) \models pd(\mathcal{M},c_1) \wedge (Q,\phi_{i+1}) \models pd(\mathcal{M},c_1) && (IH)\\
\Leftrightarrow &(Q,\phi_{i+1}) \models pd(\mathcal{M},c_1 \wedge c_2) && (Construction\ 1)
\end{aligned}$$

case 4: $c = \exists(a,c_1)$
By induction hypothesis, the claim holds for c_1 (IH).

“ $\Longrightarrow$ ”

$$
\begin{aligned}
& (Q,\phi_i) \models \exists(a,c_1) \\
\Longrightarrow\ & \exists p : (P,\iota_a^P) \hookrightarrow (Q,\phi_i) \text{ and} \\
& \exists q : (C,\iota_a^C) \hookrightarrow (Q,\phi_i) \\
& \quad \text{s.t. } q \circ a = p \text{ and } q \models c_1 && (Definition\ 5) \\
& \text{and } \phi_i \circ \overline{p} = \iota_a^P \text{ and } \phi_i \circ \overline{q} = \iota_a^C && (Definition\ 3) \\
\Longrightarrow\ & \exists \iota_d^P : P \to S_{i+1} \text{ and } \iota_d^C : C \to S_{i+1} \\
& \quad \text{s.t. } \phi_{i+1} \circ \overline{p} = \iota_d^P \text{ and } \phi_{i+1} \circ \overline{q} = \iota_d^C && (DT) \\
\Longrightarrow\ & \exists p_d : (P,\iota_d^P) \hookrightarrow (Q,\phi_{i+1}) \text{ and} \\
& \exists q_d : (C,\iota_d^C) \hookrightarrow (Q,\phi_{i+1}) \text{ and} \\
& \exists d : (P,\iota_d^P) \hookrightarrow (C,\iota_d^C) \\
& \quad \text{s.t. } q_d \circ d = p_d && (Definition\ 3) \\
& \text{and } q_d \models pd(\mathcal{M},c_1) && (IH) \\
\Longrightarrow\ & \exists d \in \mathcal{D} \text{ s.t. } (Q,\phi_{i+1}) \models d && (Construction\ 1) \\
\Longrightarrow\ & (Q,\phi_{i+1}) \models pd(\mathcal{M},\exists(a,c_1)) && (Construction\ 1)
\end{aligned}
$$

“ $\Longleftarrow$ ”

$$
\begin{aligned}
& (Q,\phi_i) \models pd(\mathcal{M},\exists(a,c_1)) \\
\Longrightarrow\ & \exists d : (P,\iota_d^P) \to (C,\iota_d^C),\ \iota_d^P : P \to S_{i+1},\ \iota_d^C : C \to S_{i+1} \text{ and} \\
& \iota_{i+1} \circ \iota_d^P = \iota_a^P \text{ and} \\
& \iota_{i+1} \circ \iota_d^C = \iota_a^C && (Construction\ 1) \\
& \text{and } \exists p_d : (P,\iota_d^P) \hookrightarrow (Q,\phi_{i+1}) \text{ and} \\
& \exists q_d : (C,\iota_d^C) \hookrightarrow (Q,\phi_{i+1}) \text{ and} \\
& \exists d : (P,\iota_d^P) \hookrightarrow (C,\iota_d^C) \\
& \quad \text{s.t. } q_d \circ d = p_d && (Definition\ 3) \\
\Longrightarrow\ & \exists p_a : (P,\iota_a^P) \hookrightarrow (Q,\phi_i) \text{ and} \\
& \exists q_a : (C,\iota_a^C) \hookrightarrow (Q,\phi_i) \text{ and} \\
& \exists a : (P,\iota_a^P) \hookrightarrow (C,\iota_a^C) \\
& \quad \text{s.t. } q_a \circ a = p_a && (DT) \\
& \text{and } q_a \models c_1 && (IH) \\
\Longrightarrow\ & (Q,\phi_i) \models \exists(a,c_1) && (Definition\ 5)
\end{aligned}
$$

□

Fig. 5.3 shows a visualisation of the final case of the proof. Typing morphisms to model level i are replaced with morphisms to model level $i+1$ such that they

commute. If c is satisfied at least one such replacement must exist and if at least one such replacement exists c must be satisfied.

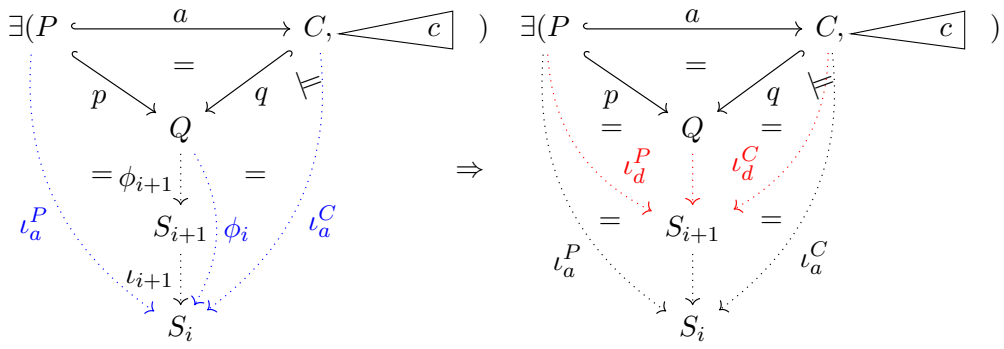


Figure 5.3.: Visualisation of proof construction for Theorem 2.

We now use this construction to replace all constraints in a model by constraints on the lowest level. We call this operation **bottom propagation**.

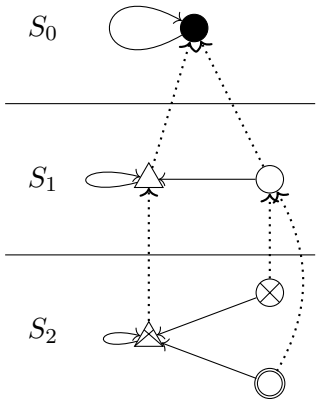
Algorithm 1 (Bottom Propagation)**.** Given a model $\mathcal{M} = (\mathcal{S},Sig,Spec)$ with type graphs $\mathcal{S} = [(S_0,\iota_{S_0}),\ldots,(S_n,\iota_{S_n})]$, signatures $[\Sigma_0,\ldots,\Sigma_n]$ and specifications $[\mathfrak{S}_0,\ldots,\mathfrak{S}_n]$, the bottom propagated model $bp(\mathcal{M}) = \mathcal{M}_B$ is constructed as follows:

```
input: M
output: bp(M)
begin
  M_B ← M

  for i in 0 to n − 1 do
    foreach c in 𝔖_i do
      M_B ← cr(M_B,c)
    end
  end

  return M_B
end
```

This algorithm propagates constraints down from top to bottom. It is guaranteed to be terminating because the amount of constraints on the layer currently operated on is always decreasing, layers are followed strictly sequentially and each constraint on layer i only creates finitely many more constraints on $i+1$.

Example 21. In Fig. 5.4 a meta-model with two initial constraints is given. By c_1 a node with an outgoing edge must exist. By c_2 a triangle node with a loop must exist. Each loop iteration of the algorithm results in constraints of one level being replaced on lower levels from the top down. From the initial state I constraints are propagated down in the loop steps L_0 and L_1. Note that the second initial constraint does not change in L_0 because it is not defined on S_0.

Observation 3. Since typing on each layer is done via a graph morphism, each element is mapped to exactly one parent. The inverted typing edges form a tree.



Step	c_1	c_2
I	$\exists$(● ⟶ ●)	$\exists$(△)
L_0	$\exists$(○ ⟶ △) $\vee$ $\exists$(△ ⟶ △)	$\exists$(△)
L_1	$\exists$(◎ ⟶ △) $\vee$ $\exists$(△ ⟶ △) $\vee$ $\exists$(⊗ ⟶ △)	$\exists$(△)

Figure 5.4.: Bottom propagation for a three layer model with two initial constraints

This means that two different typing paths can not result in the same element on the lowest level and disjoint constraints must remain disjoint under bottom propagation.

Theorem 3 (Equivalence of models under bottom propagation)**.** Given a model $\mathcal{M} = (\mathcal{S}, Sig, Spec)$ with signatures $[\Sigma_0, \ldots, \Sigma_n]$ and specifications $[\mathfrak{S}_0, \ldots, \mathfrak{S}_n]$, Algorithm 1 yields an equivalent model $\mathcal{M}_B$ with $\forall i < n, \mathfrak{S}_i = \emptyset$.

Proof (Theorem 3)**.** By Theorem 2, every model changing step in Algorithm 1 consists of an equivalent transformation. Thus the final output of the algorithm must be equivalent to $\mathcal{M}$. Each $pd(\mathcal{M}, c)$ removes c from the specification on its level and adds no new constraints to it. Since the algorithm is applied from highest to lowest level and to each constraint per level this means all specifications $\mathfrak{S}_i$ with $i > n$ must be empty. □

5.2. MR Repair Approach

A model usually consists of multiple constraints, potentially on different model levels, that should be fulfilled. For such a conjunction of constraints one approach to constructing a repair program is:

Procedure 1 (Divide and combine)**.**

1. Propagate all constraints to the lowest level with Algorithm 1.
2. Separate the model constraints by (possibly implicit) conjunctions.
3. Construct repair programs for each separate constraint by choosing one of these approaches

a) Define applicable rule sets and construct repair programs from rule sets [HS19]

b) Construct repair programs from constraints and adapt them according to modeling needs [HS18]

c) Construct a repair program manually

4. Compose repair programs to a model repair program if possible.

The main idea is to first transform the model specification to a set of constraints on the same layer. This removes the need for separate handling of diverging subtypes.

Then repair programs for each sub-constraint are constructed, assuming that this is easier than finding a repair program for the complete model specification directly. Different model constraints are semantically equivalent to the constraint formed by their conjunction because all constraints must hold for a model to be satisfied. Since propagated constraints can form large conjunctions as well we may treat them as separate constraints for the model, breaking them along top level conjunctions.

Finally an analysis of constraint preservation is performed hopefully yielding a repairing execution order of sub-programs.

Note. This approach will only find repair programs of a very specific structure. It is not exhaustive but rather a general guide to finding a repair program. Repair programs being preserving for another's constraint is not always necessary. In [HS18] a set of criteria is given under which a repair program may be constructed without preservation. Additionally it is shown that terminating repair programs can be constructed for proper conditions, that is conditions with alternating quantifiers ending with $true$ or of the form $\exists(A,\nexists C)$. If the bottom propagated model constraint can be transformed into a proper condition and predefining rule sets is not a concern this approach can be used to construct a model repair program.

Theorem 4 (Divide and combine procedure yields a repair program)**.** For satisfiable models where an execution order of repair programs for each constraint exists such that each program is preserving for constraints of previous programs, the steps in Procedure 1 yield a model repair program.

For the proof we use the fact that a repair program for a given model is also a repair program for any equivalent model.

Lemma 1 (Repair Program Equivalence)**.** Repair programs for a model $\mathcal{M}$ also repair equivalent models $\mathcal{M}^* \equiv \mathcal{M}$.

Proof (Lemma 1)**.** A program P is a repair program for $\mathcal{M}_1$ if it transforms all candidate instances into valid instances (Definition 19). Two models $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent if they are defined on the same model stack and every typed graph that is an instance of $\mathcal{M}_1$ is an instance of $\mathcal{M}_2$ and vice versa (Definition 18). Thus any repair program for $\mathcal{M}_1$ is also a repair program for $\mathcal{M}_2$ and vice versa. □

Proof (Theorem 4)**.** Given a model $\mathcal{M} = (\mathcal{S}, Sig, Spec)$ with signatures $[\Sigma_0, \ldots, \Sigma_n]$ and specifications $[\mathfrak{S}_0, \ldots, \mathfrak{S}_n]$, step 1 of the procedure yields an equivalent model $\mathcal{M}^*$ with all constraints on level n by (Theorem 3). The repair programs constructed in step 2 are correct for their individual constraints [HS18, HS19]. By definition the model is satisfiable and an execution order of created programs exists, such that each program is preserving for the constraint of the previous program. Thus it is possible to compose them to a full repair program. Since $\mathcal{M}^*$ is equivalent to $\mathcal{M}$ the repair program also repairs $\mathcal{M}$ by Lemma 1. □

Note. The main issue here is that the simple sequential composition of repair programs $\langle P_1; P_2 \rangle$ is a model repair program only if P_2 preserves the constraint fulfilled by P_1. Thus the challenge becomes to identify which of the programs preserve constraints of others and finding a sequence that actually guarantees a satisfied model. In general it is not necessary for a repair program to have that structure. Every individual sub-program may be neither guaranteeing nor preserving for a given constraint but the whole may still be a repair program.

Example 22. Consider the example given in Fig. 5.5: Here the different coloured nodes represent differently typed objects. A grey colour signifies an object of arbitrary type. None of the individual rules guarantee or preserve the constraint that there exists one and only one triangle of these objects. Nevertheless the program is a repair program.

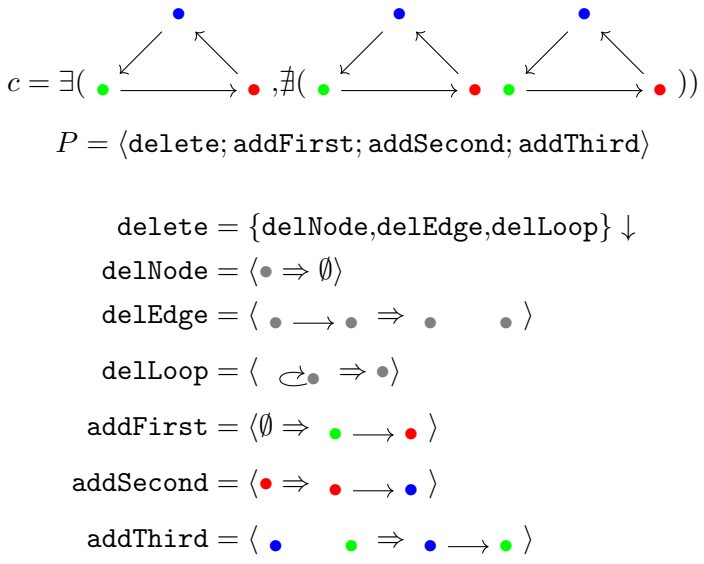


Figure 5.5.: Repair Program for triangle constraint without preservation or guarantee on subprograms.

It is easier to reason that a program guarantees a constraint when information about preservation of sub-programs can be provided. To achieve this in general an

algorithm would be necessary that decided whether a given program or even single transformation rule is preserving for a constraint.

Theorem 5 (Undecidability of constraint guarantee for rules)**.** For a given constraint d and a rule ϱ it is undecidable in general whether ϱ is guaranteeing for d.

Proof (Theorem 5)**.** Consider the empty rule with some application condition c:

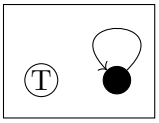
$$\langle \emptyset \hookleftarrow \emptyset \hookrightarrow \emptyset, c \rangle$$

The underlying plain rule will leave any graph G it is applied to unchanged. Thus the question becomes whether $G \models d$ for all graphs $G \models c$. This is the case if and only if $c \implies d$. The implication problem for graph conditions is undecidable [Pen09]. Thus the guarantee problem is undecidable. □

Theorem 6 (Undecidability of constraint preservation for programs)**.** For a given typed constraint d and a typed graph program P it is undecidable in general whether P is preserving for d.

We can show this by using the non-determinism of graph programs to construct an arbitrary output graph given any input. For this program to be preserving for a condition it must be a tautology or unsatisfiable.

Proof (Theorem 6)**.** In the following let a graph program P and a condition d be typed on

$TG =$ 

such that no element in c is is of type Ⓣ . Now consider the following program:

$$P = \langle \langle \texttt{delGraph}; \texttt{modify} \downarrow \rangle; \texttt{delTerm} \rangle$$

$\texttt{addNode} = \langle \emptyset \Rightarrow \bullet \rangle$

$\texttt{delNode} = \langle \bullet \Rightarrow \emptyset \rangle$

$\texttt{addEdge} = \langle \bullet \quad \bullet \Rightarrow \bullet \longrightarrow \bullet \rangle$

$\texttt{delEdge} = \langle \bullet \longrightarrow \bullet \Rightarrow \bullet \quad \bullet \rangle$

$\texttt{addLoop} = \langle \bullet \Rightarrow \bullet \circlearrowleft \rangle$

$\texttt{delLoop} = \langle \bullet \circlearrowleft \Rightarrow \bullet \rangle$

$\texttt{addTerm} = \langle \emptyset \Rightarrow \text{Ⓣ} \rangle$

$\texttt{delTerm} = \langle \text{Ⓣ} \Rightarrow \emptyset \rangle$

$\texttt{hasTerm} = \langle \emptyset \Rightarrow \emptyset, \exists(\text{Ⓣ}) \rangle$

$\texttt{stop} = \langle \emptyset \Rightarrow \emptyset, false \rangle$

$$\texttt{delGraph} = \{\texttt{delNode}, \texttt{delEdge}, \texttt{delLoop}\} \downarrow$$

$$\texttt{addElement} = \{\texttt{addNode}, \texttt{addEdge}, \texttt{addLoop}, \texttt{addTerm}\}$$

$$\texttt{modify} = \texttt{if hasTerm then stop else addElement}$$

For any input graph this program will output an arbitrary, possibly empty, graph. Thus it is preserving for d if and only if:

$$(d \Leftrightarrow true) \vee (d \Leftrightarrow false)$$

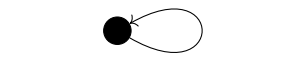
That is if d is a tautology or unsatisfiable. If preservation was decidable for any c then if the program is preserving for d we could evaluate the condition on any graph and if the condition holds it is a tautology and unsatisfiable otherwise. The tautology and satisfiability problems for graph conditions are undecidable [Pen09]. Thus the preservation problem for typed graph programs is undecidable. □

Note. The reason we require typing in this theorem is to mark the termination node. It can therefore be adapted directly to graphs having other means of marking like labeled or attributed graphs. The program used in the proof is very different from the ones we would construct for the repair of any constraint. There are various conditions under which a transformation rule and by extension a graph program can be decided to be preserving for a given condition.

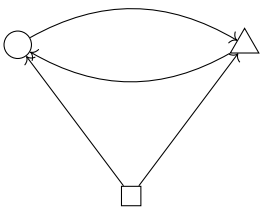
Observation 4. A typed transformation rule ϱ is preserving for a constraint c typed by the same graph if:

- The types of elements in ϱ and c are disjoint or
- ϱ is non-deleting and c is of the form $\exists(a,true)$ or
- ϱ is purely deleting and c is of the form $\nexists(a,true)$ or
- c is a tautology or unsatisfiable.

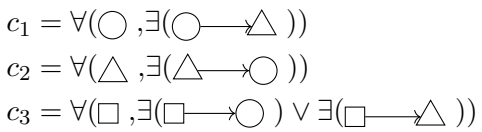
Example 23 (Repair Construction)**.** We construct a repair program for the pull-down example.



S_0



S_1



We attempt to construct repair programs from rule sets. Transformations and programs with interface and context can be introduced to type graphs as in [HS19]. The rules `Sel`(x,ac) and `Uns`(y) denote the selection and unselection of elements.

For a rule set to be usable for repair program construction there must exist a *ac-resulting* transformation with it for the given condition. This means it preserves its match and the result satisfies the condition.

All of our constraints are of the form $\forall(a,c)$. Thus their repair programs will be of the form $\langle \texttt{Sel}(a,\neg c)); P_c; \texttt{Uns}(a)\rangle \downarrow$. Here P_c is the repair program for c. For

$$d_1 = \exists(\bigcirc \hookrightarrow \bigcirc \longrightarrow \not\triangle\ )$$

we use the transformation rule $r_1 = \langle \bigcirc \Rightarrow \bigcirc \longrightarrow \not\triangle\ \rangle$. There exists a d_1-resulting transformation with r_1:

$$\langle \bigcirc \Rightarrow \bigcirc \longrightarrow \not\triangle\ , \not\exists(\bigcirc \longrightarrow \not\triangle\ )\rangle \downarrow$$

Thus our first repair program is:

$$P_1 = \langle \texttt{Sel}(\bigcirc\ , \not\exists(\bigcirc \longrightarrow \not\triangle\ )); r_1; \texttt{Uns}(\bigcirc\ )\rangle \downarrow$$

Similarly we construct P_2 via $r_2 = \langle \triangle \Rightarrow \triangle \rightleftarrows \bigcirc\ \rangle$:

$$P_2 = \langle \texttt{Sel}(\triangle\ , \not\exists(\triangle \longrightarrow \bigcirc\ )); r_2; \texttt{Uns}(\triangle\ )\rangle \downarrow$$

c_3 contains a disjunction and is not directly covered by the constructions in [HS19]. They are briefly covered in [HS18]. In our case we simply decide which edge should be added to a square if none is present and manually construct the program using the following rules:

$$r_{3a} = \langle \square \quad \bigcirc \Rightarrow \square \longrightarrow \bigcirc\ \rangle$$
$$r_{3b} = \langle \square \Rightarrow \square \longrightarrow \bigcirc\ \rangle$$

We combine these to repair program P_3:

$$P_3 = \langle \texttt{Sel}(\square\ , \neg(\exists(\square \longrightarrow \bigcirc) \vee \exists(\square \longrightarrow \not\triangle)); \texttt{try}\ r_{3a}\ \texttt{then skip else}\ r_{3b}; \texttt{Uns}(\square)\rangle \downarrow$$

Next we investigate which program is preserving for the other constraints to find a possible execution order for a model repair program. This is shown in Fig. 5.6. P_1 is preserving for c_3 as it may not create a square node without an outgoing edge. It is not preserving for c_2 because it can create a triangle node without an outgoing edge. P_2 is preserving for both c_1 and c_2 because it does not create any square nodes and any circle nodes it creates will have an outgoing edge. P_3 is preserving for c_2 as it never creates a triangle node but may break c_1 by creating one.
This means that P_1 must be run before P_2. P_3 must be run before P_1. Thus a preserving execution order exists and $P_R = \langle P_3; P_1; P_2\rangle$ is a repair program for the model.

	c_1	c_2	c_3
P_1	-	✗	✓
P_2	✓	-	✓
P_3	✗	✓	-

✓ left program preserves top constraint
✗ otherwise

Figure 5.6.: Preservation analysis for repair construction

Note. The repair programs constructed by the methods above guarantee a valid model instance but may add or remove elements needlessly when run on a (partially) valid model. Thus it is advisable to first check if any constraints are violated before attempting any repairs.

The use of graph conditions makes finding inconsistencies easier than in DPF. Any candidate instance must be fully typed by the model. For each level the instance graphs elements can be replaced by the targets of the corresponding type morphism. The resulting graph is a typed graph on the model level and the graph conditions on that level can be checked directly. For constraints that have no intersection running only the corresponding partial repair program may be sufficient. This is always the case if they are preserving for the other constraints.

5.3. Evaluation

This section highlights the differences between DPF and MR and their respective advantages and disadvantages. Since both frameworks use similar terminology the first part provides the distinctions between them.

Predicate/Template/Signature

In DPF predicates are given by a shape and a set of valid instances that may be expressed in any way. This allows for the definition of any constraint. However, it can make automated checking of instance validity challenging with rising constraint complexity. MR templates are graph conditions and thus limited to first-order expressiveness. This makes checking model validity decidable in general. Both templates and predicates are instantiated by typing them on a graph. Sets of templates or predicates are called signatures in both frameworks.

Constraint/Specification/Model/Instance

The instantiation of a predicate or template is referred to as a constraint in both frameworks. They both serve the purpose of mapping a predefined template/predicate to a given model. Sets of constraints of the same signature are called specifications when applied to the same graph.

An instance in DPF refers to an instance of a single specification. This means that the instance term here is limited to constraints on a single stack level.

In MR the term model is separately defined as a stack of type graphs with corresponding specifications. An instance is defined with regards to a model and thus a valid instance must conform to all model specifications. This does not provide an increase in expressiveness (see Theorem 3) but simplifies modeling higher level constraints.

Advantages of MR

All constraints in MR are typed graph conditions. Thus whether a given graph satisfies the model is always decidable by checking the satisfaction of the conjunction of model constraints. The uniform semantics means that this conjunction is again a typed graph condition.

Model and model equivalence are defined in MR and constructions to pull all constraints to the lowest model level without changing its semantics exist by Theorem 3. This means existing graph repair approaches can more easily be adapted to MR because no special handling of high level constraints is necessary.

The modular inclusion of constraints into the model semantics means that graph conditions can easily be replaced by more expressive constraints as necessary. This merely requires the constraints to be typable on the model. Examples of how this may be done are provided in Appendix A.2.

Disadvantages of MR

The model framework is built on a stack of typed graphs. Because of this, every layer is required to be fully and uniquely typed by the previous one. In some cases this means an element must be included again on a lower level despite not containing any real extra information. One example of this is the ID-field in the running example in Fig. 4.1.

A similar notion applies to constraints. Occasionally a constraint may quantify over all elements of a type on a higher level and then express a restriction on a lower level. In this case a full conjunction of the sub-types is necessary to express the constraint. An example of this arises with the control and data flow graphs in Section 4.3.

To achieve repairablity constraints have been limited to first-order logic. This means not all common constraints in Section 4.4 can be modeled. This limitation is also visible in the case studies.

The constraint propagation used to apply existing graph repair approaches can result in very large conditions as every possible sub-type results an extra part. Additionally each constraint is checked for preservation of all other constraints. This means the process could become very slow and may not find a repair program even if one exists.

Combining MR and DPF

Every model expressible in MR is also expressible in DPF. This expressive power may be combined with the formal semantics and repairability of MR by defining first-order properties by graph conditions where possible in a DPF model. Then repair could be established for this first-order sub-model. This means the more expressive but more informal semantics of predicates could be limited to where they are actually necessary. The sub-model repair program could then possibly be adapted to a model repair program by manually accounting for the extra constraints.

6. Related Work

Rabbi et al. [RLYK15] introduce a model completion approach for DPF models. For every predicate used in a model, a set of completion rules is introduced. These rules consist of a double pushout transformation rule with a negative application condition given by $N \hookleftarrow L \hookleftarrow K \hookrightarrow R$. Each graph in the rule is typed by the arity of the predicate. A rule may only be applied if the pattern N is not present in the match. These rules can be applied repeatedly until a model conforms to the specification. A set of sufficient criteria for the execution order to guarantee termination is provided.

In [RMKL18] Rabbi et al. provide an algorithm to check whether a model transformation rule is conformance preserving. That is whether applying the rule to any valid model is guaranteed to yield valid model again. This approach uses a set of sufficient conditions for conformance preservation. They distinguish between conformance by type (i.e. whether every type of the instance is valid) and conformance by satisfaction of constraints. A graph transformation rule may include multiple negative application conditions here. The set of allowed constraints is limited to $\forall(L,\exists(R))$ and $\forall(L,\nexists(R))$ with graphs L and R. An extension of the algorithm to nested graph constraints is listed as future work.

In [HS18] a set of constructions are introduced that yield repair programs for a subset of satisfiable graph constraints. This is done by constructing sets of transformation rules for each atomic constraint. All constructed programs are maximally preserving. Since graph programs are non-deterministic this means that for a given program there exists a computation path yielding a graph that fulfils the constraint such that a minimal amount of nodes or edges is deleted.

[HS19] allows the construction of constraint repair programs from a given rule set. These programs are applicable to arbitrarily nested proper conditions. These are conditions with alternating quantifiers ending on $\exists b$ or is of the form $\exists(a,\nexists b)$. Conjunctions and disjunctions are not covered. Parts of nested conditions are then connected on the same match. To achieve this transformation, rules are extended with interfaces and context. These can enforce that a selected subgraph is kept intact over the transformation. Thus any rule may guarantee that a match exists for the following. Similarly the graph programs with interfaces are introduced such that input and output must match a certain pattern. Repair programs can be constructed if the given rule set is compatible with the proper condition. Checking compatibility is undecidable for non-deleting rule sets and positive constraints and semi-decidable for arbitrary rule sets and constraints.

Nassar et al. [NRA17, NKR17] provide an algorithm for repairing multiplicity constraints in EMF-Models. They achieve this by first deleting all super-numerous

elements in the instance and then adding in missing elements as necessary. The algorithm is proven to be correct. For MR models consisting only of multiplicity constraints this algorithm can be directly adapted. Since it does not differentiate between model layers constraints above level n must first be propagated to the lowest level by Algorithm 1.

Navarro et al. [NOPL16] extend nested graph conditions with paths. They provide a sound and complete tableau method for reasoning about path properties. This is done by extending graphs to graph patterns, allowing special edges that signify a path between nodes. Formulas may include arbitrary (not just injective) morphisms. They generalize this work in [LNOP18] to be applicable to different classes of graphs. Additionally, they provide a set of inference rules for reasoning in this logic. This approach is applicable to attributed typed graphs.

Poskitt and Plump [PP14] introduce an extension of nested conditions to monadic second-order properties. This is achieved by using variables and corresponding interpretation conditions to reason about sets of node and edge variables. They define a weakest precondition system to verify correctness of graph programs for these conditions. A further explanation and examples for a possible adaptation into MR is provided in Appendix A.2.

Flick [Fli16] introduces recursively nested graph conditions called μ-conditions to express non-local properties. These have the same expressiveness as fixed point extensions to first-order logic on finite graphs and are not equivalent to M-conditions. This is achieved by allowing placeholders in nested conditions which may be substituted by further nested conditions which themselves may contain placeholders.

Radke [Rad13] presents an extension of graph conditions called HR^* graph conditions in his PhD thesis. These use hyperedges together with hyperedge replacement grammars to express properties up to counting monadic second-order. A further explanation and examples for a possible adaptation into MR is provided in Appendix A.2.

Schneider et al. [SLO19] introduce an incremental graph repair approach generating a complete set of least-changing repair operations. That is a set of direct transformations for the given instance such that each has no repairing subtransformation. The choice of which repair to apply can then be made by the user. This approach is applied to a given instance of the model and does not generate general repair programs.

Attie et al. [ACD+15] propose a model repair approach for finite Kripke structures relative to a CTL formula. This is done by looking for a sub-structure that satisfies the condition and deleting excess elements. They provide an implementation of this as an interactive graphical tool.

Faland et. al. [FvdA15] study the problem of fixing an existing model to better represent the reality of a given process. This is done by gathering event logs and checking for differences between modeled and actual behaviour. They propose to automatically repair a model (BPMN, Petri net, etc.) given a log. That is, altering the model so that it can replay the log correctly. They provide an implementation and validate the system on real-life event logs. This is likely a sensible approach,

when trying to model an existing process.

Taentzer et al. [TOLR17] deal with the problem of model inconsistencies arising during the modeling process. Changes may lead to temporary model violations but must ultimately be resolved. They do this by identifying edit-operations. Models are then repaired by applying complement edit-operations while preserving the changes made. They show that this approach is sound and provide a prototype implementation.

7. Conclusion

In this thesis we introduce a constraint based modeling framework MR inspired by the Diagram Predicate Framework (DPF). We replace DPF predicates with graph conditions to make instance validity decidable in general and open approaches to model repair.
We establish that

1. satisfaction of MR models is checkable (Fact 3) and
2. model equivalence is undecidable (Theorem 1).

A set of constructions to repair these models is provided:

1. Constraints in MR models can be replaced by constraints on a lower level such that the model remains equivalent (Theorem 2). This construction can be applied to all model constraints sequentially by level to yield an equivalent model where all constraints are on the lowest level (Theorem 3).
2. We use this fact to propose a repair approach that yields a model repair program if there exists a preserving execution order of constraint repair programs for each model constraint (Theorem 4).
3. We show that the problem of deciding that a given typed graph program is preserving for a given constraint is undecidable (Theorem 6) and prove the same for a transformation rule guaranteeing a constraint (Theorem 5).

Finally, we also identify a set of common constraints in software engineering which may provide a guideline for future extensions of constraint expressiveness.

In the future we would like to provide an implementation of the framework to test its actual usefulness in modeling. More expressive constraints would be helpful for this. M-conditions and and HR*-conditions seem to be good candidates for this. Models may also be extended with attributes and labels to easier model software properties like ranges of variables.

In the vein of [NRA17] more subsets of constraints could be identified for which effective repair algorithms exist.

The question under which conditions preservation for transformation rules and graph programs is decidable could be further investigated. If an algorithm for conditions up to a certain complexity (i.e. a certain nesting depth or structure) could be provided, it would make repairability of models with only these constraints decidable.

If models are to incorporate more expressive constraints, repair for monadic second-order and higher properties has to also be researched.

A. Appendix

A.1. Category Theory

Category theory can be used to abstract various mathematical objects into categories with common properties. A theorem proved for categories in general can then be applied to each individual category alleviating the need to essentially prove the same thing multiple times. By defining graphs as a category the authors can make use of already established properties.

In [EEPT06] Ehrig et al. give the following definition:

Definition 20 (category)**.** A **category 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;
- $\forall A,B,C \in Ob_C$ a composition operation $\circ(A,B,C) : Mor_C(B,C) \times Mor_C(A,B) \to Mor_C(A,C)$;
- for each object $A \in Ob_C$ an identity morphism $id_A \in Mor_C(A,A)$ such that
  - 1. Associativity. For all objects A,B,C,$D \in Ob_C$ and morphisms $f : A \to B$, $g : B \to C$, $h : C \to D$ 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 $f : A \to B$ it holds that $f \circ id_A = f$ and $id_B \circ f = f$

Definition 21 (commutation)**.** In category theory a diagram (or path of morphisms) is **commutes** if every path with the same start and endpoint leads to the same result.

Consider the example in Fig. A.1 showing three morphisms ϕ_i and three graphs A,B,C. The paths $\phi_2 \circ \phi_1$ and ϕ_3 both lead from A to C and thus commute if $\phi_2 \circ \phi_1 = \phi_3$.

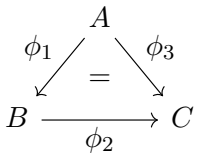


Figure A.1.: Example of a commuting diagram.

Pullback and Pushout

Commutation is an important property of two complementary constructions: **pullback** and **pushout** as defined by Ehrig et al. in [EEPT06] and displayed in A.2:

Definition 22 (pushout)**.** Given two morphisms $f : A \to B$ and $g : A \to C$ a **pushout** (D,α,β) is defined by a pushout object D and morphisms $\alpha : C \to D$ and $\beta : B \to D$ with $\alpha \circ f = \beta \circ g$, such that this universal property is fulfilled: For all objects D' with morphisms $\alpha' : C \to D'$ and $\beta' : B \to D'$ with $\alpha' \circ f = \beta' \circ g$ there is a **unique** morphism $\theta : D \to D'$ such that $\theta \circ \beta = \beta'$ and $\theta \circ \alpha = \alpha'$.

This universal property ensures that the pushout object is "minimal" in the sense, that there may be no other object with its properties with a morphism into it (otherwise it would be the pushout itself). Ehrig et al. also give the intuition of the pushout being the result of gluing two objects along a common subobject.

Definition 23 (pullback)**.** Given two morphisms $f : C \to D$ and $g : B \to D$ a **pullback** (A,α,β) is defined by a pullback object A and morphisms $\alpha : A \to B$ and $\beta : A \to C$ with $\alpha \circ g = \beta \circ f$ and the universal property: For all objects A' with morphisms $\alpha' : A' \to B$ and $\beta' : A' \to C$ and $\alpha' \circ g = \beta' \circ f$, there is a **unique** morphism $\theta : A' \to A$ such that $\alpha \circ \theta = \alpha'$ and $\beta \circ \theta = \beta'$.

The intuition here is that the pullback is a generalized intersection of two objects over a common third object. The universal property ensures that the pullback object is "maximal" in the sense every other object with these properties has a morphism into it.

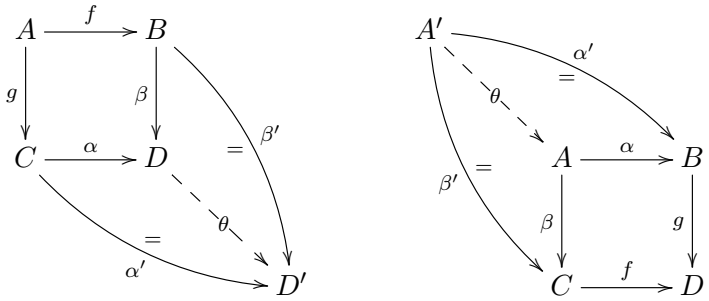


Figure A.2.: Visualisation of pushout (left) and pullback (right) as commuting diagrams.

A.2. More Expressive Constraints

In this section we include more expressive constraints in place of graph conditions into our model definition. To achieve this we show how the conditions may be typed by the model.

M-conditions

M-conditions [PP14] are an extension of graph conditions with variables to make them equivalently expressive to monadic second-order formulas. To achieve this they introduce sets of node and edge variables `VSetVar` and `ESetVar`. They also use a predicate $path_G(v,w,E)$ to denote a path between vertices v,w in a graph G without use of the edges $e \in E$.

Definition 24 (interpretation, interpretation condition)**.** An **interpretation** I in a graph G is a partial function $I : \texttt{VSetVar} \cup \texttt{ESetVar} \to 2^{V_G} \cup 2^{E_G}$, assigning sets of edges or nodes to edge or node variables respectively. An **interpretation constraint** is a boolean expression that may be derived from the following grammar:

Constraint ::=`Vertex '∈' VSetVar`|`Edge '∈' ESetVar`
| `path '(' Vertex ',' Vertex ['` `,' not Edge {'|' Edge}] ')'`
| `not` Constraint| Constraint`(and | or)` Constraint | `true`

Given a constraint γ, an interpretation I in G and a morphism q with codomain G, the value of $\gamma^{I,q}$ is defined inductively. If γ contains a set variable for which I is undefined, then $\gamma^{I,q} = false$. Otherwise, if γ is `true`, then $\gamma^{I,q} = true$. If γ is of the form $x \in \texttt{X}$ with x a node or edge identifier and `X` a set variable, then $\gamma^{I,q} = true$ if $q(x) \in I(\texttt{X})$. If γ has the form `path`(v,w) with v,w node identifiers, then $\gamma^{I,q} = true$ if the predicate $path_G(q(v),q(w),\emptyset)$ holds. Respectively `path`$(v,w,$`not`$e_1|\ldots|e_n) = true$ if $path_G(q(v),q(w),\{q(e_1),\ldots,q(e_n)\})$ holds. If γ has the form `not` γ_1 then $\gamma^{I,q} = true$ if $\gamma_1^{I,q} = false$. If γ has the form γ_1 `and` γ_2 then $\gamma^{I,q} = true$ if $\gamma_1^{I,q} = true$ and $\gamma_2^{I,q} = true$.

These conditions are used to assign node variables to certain sets. The `path` construct denotes the existence of a path between two vertices without use of the edges listed. This is not necessary for expressiveness but a useful shorthand notation.

Example 24. The following interpretation condition holds on an interpretation if there is a path from $\texttt{v}_1$ to $\texttt{v}_2$ without $\texttt{e}_1$ and the vertices are assigned to different variables.

$$\gamma = \texttt{path}(\texttt{v}_1,\texttt{v}_2,\texttt{not e}_1)\ \texttt{and}\ \texttt{v}_1 \in \texttt{X}\ \texttt{and}\ \texttt{v}_2 \in \texttt{Y}$$

Definition 25 (M-condition)**.** An **M-condition** over a graph P is of the form `true`, $\exists_{\texttt{V}}\texttt{X}[c]$, $\exists_{\texttt{E}}\texttt{X}[c]$ or $\exists(a|\gamma,c')$, where $\texttt{X} \in$ `VSetVar` (resp. `ESetVar`), c is an M-condition over P, $a : P \hookrightarrow C$ is an injective morphism, γ is an interpretation constraint over items in C and c' is an M-condition over C. Boolean formulas over M-conditions are also M-conditions.

A morphism $p : P \hookrightarrow G$ satisfies a condition c over P with respect to an interpretation I ($p \models^I c$) inductively. If $c =$ `true`, then $p \models^I c$. If $c = \exists_{\texttt{V}}\texttt{X}[c]$ (resp. $c = \exists_{\texttt{E}}\texttt{X}[c]$), then $p \models^I c$ if $p \models^{I'} c'$ where $I' = I \cup \{\texttt{X} \mapsto V\}$ for some $V \subseteq V_G$ (resp.

$\{\mathtt{X} \mapsto E\}$ for some $E \subseteq E_G$). If c has the form $\exists(a|\gamma,c')$, then $p \models^I c$ if there is an injective morphism $q : C \hookrightarrow G$ such taht $q \circ a = p, \gamma^{I,q} = true$ and $q \models^I c'$.

A graph G satisfies an M-constraint $G \models c$ if $i_G : \emptyset \hookrightarrow G \models^{I_\emptyset} c$, where $I_\emptyset$ is the empty interpretation in G.

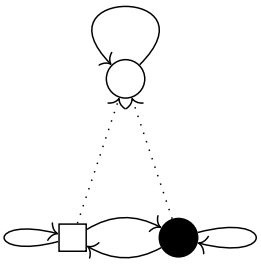
Example 25. The following M-constraint expresses that a graph can be separated into two subgraphs such that each subgraph is fully connected and there is no connection between subgraphs.

$$\gamma_{same} = (\mathtt{v} \in \mathtt{X} \text{ and } \mathtt{w} \in \mathtt{X}) \text{ or } (\mathtt{v} \in \mathtt{Y} \text{ and } \mathtt{w} \in \mathtt{Y})$$

$$\begin{aligned}\exists_{\mathtt{V}}\mathtt{X},\mathtt{Y}[&\forall(\bullet_{\mathtt{v}},\exists(\bullet_{\mathtt{v}}|(\mathtt{v} \in \mathtt{X} \text{ or } \mathtt{v} \in \mathtt{Y}) \text{ and not } (\mathtt{v} \in \mathtt{X} \text{ and } \mathtt{v} \in \mathtt{Y})))\\ &\wedge \forall(\bullet_{\mathtt{v}}\,\bullet_{\mathtt{w}},\exists(\bullet_{\mathtt{v}}\,\bullet_{\mathtt{w}}\,|\gamma_{same}) \Leftrightarrow \exists(\bullet_{\mathtt{v}} \to \bullet_{\mathtt{w}}))]\end{aligned}$$

M-conditions can be used on type graphs by the same extension used in Definition 4, replacing all regular graph morphisms with typed graph morphisms. Construction 1 is adaptable to these constraints. The variables in the used interpretation constraints are not typed as they just refer to already typed elements of a graph.

Example 26. The following shows a meta model with the derivation of the previous constraint to its sub-types.



$$\gamma_{assigned} = (\mathtt{v} \in \mathtt{X} \text{ or } \mathtt{v} \in \mathtt{Y}) \text{ and not } (\mathtt{v} \in \mathtt{X} \text{ and } \mathtt{v} \in \mathtt{Y})$$
$$\gamma_{same} = (\mathtt{v} \in \mathtt{X} \text{ and } \mathtt{w} \in \mathtt{X}) \text{ or } (\mathtt{v} \in \mathtt{Y} \text{ and } \mathtt{w} \in \mathtt{Y})$$

$$\begin{aligned}c = \exists_{\mathtt{V}}\mathtt{X},\mathtt{Y}[&\forall(\circ_{\mathtt{v}},\exists(\circ_{\mathtt{v}}|\gamma_{assigned}))\\ &\wedge \forall(\circ_{\mathtt{v}}\,\circ_{\mathtt{w}},\exists(\circ_{\mathtt{v}}\,\circ_{\mathtt{w}}\,|\gamma_{same}) \Leftrightarrow \exists(\circ_{\mathtt{v}} \to \circ_{\mathtt{w}}))]\\ d = \exists_{\mathtt{V}}\mathtt{X},\mathtt{Y}[&\forall(\bullet_{\mathtt{v}},\exists(\bullet_{\mathtt{v}}|\gamma_{assigned})) \wedge \forall(\square_{\mathtt{v}},\exists(\square_{\mathtt{v}}|\gamma_{assigned}))\\ &\wedge \forall(\bullet_{\mathtt{v}}\,\bullet_{\mathtt{w}},\exists(\bullet_{\mathtt{v}}\,\bullet_{\mathtt{w}}\,|\gamma_{same}) \Leftrightarrow \exists(\bullet_{\mathtt{v}} \to \bullet_{\mathtt{w}}))\\ &\wedge \forall(\square_{\mathtt{v}}\square_{\mathtt{w}},\exists(\square_{\mathtt{v}}\square_{\mathtt{w}}|\gamma_{same}) \Leftrightarrow \exists(\square_{\mathtt{v}} \to \square_{\mathtt{w}}))\\ &\wedge \forall(\bullet_{\mathtt{v}}\square_{\mathtt{w}},\exists(\bullet_{\mathtt{v}}\square_{\mathtt{w}}|\gamma_{same}) \Leftrightarrow \exists(\bullet_{\mathtt{v}} \to \square_{\mathtt{w}}))\\ &\wedge \forall(\square_{\mathtt{v}}\,\bullet_{\mathtt{w}},\exists(\square_{\mathtt{v}}\,\bullet_{\mathtt{w}}\,|\gamma_{same}) \Leftrightarrow \exists(\square_{\mathtt{v}} \to \bullet_{\mathtt{w}}))]\end{aligned}$$

HR* graph conditions

In [Rad13] an extension to graph conditions called HR* graph conditions is introduced. These allow the use of hyperedges together with a context free edge

replacement grammar in a conditions graphs. This allows for the definition of non-local graph properties, most importantly path and loop conditions. They can also describe node counting properties such as "there exists an even number of nodes", expressing a true superset of MSO-properties.

Example 27. The following HR* graph condition expresses the existence of a path between two nodes.

$$\exists(\bullet_1 \xrightarrow{+} \bullet_2) \text{ with } \bullet_1 \xrightarrow{+} \bullet_2 ::= \bullet_1 \to \bullet_2 | \bullet_1 \to \bullet \xrightarrow{+} \bullet_2$$

Additionally a new form of condition $\exists(P \sqsupseteq C,c)$ is introduced. Intuitively this allows the condition c to be applied to a subgraph C of P **after** its contained grammars are resolved.

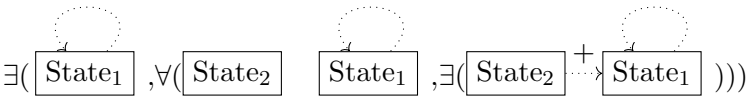
Example 28. The following HR* graph condition that there is a path from node 1 to node 2 and every node along that path has an additional outgoing edge.

$$\exists(\bullet_1 \xrightarrow{+} \bullet_2, \forall(\bullet_1 \xrightarrow{+} \bullet_2 \sqsupseteq \bullet_3, \exists(\bullet_3 \to \bullet))) \text{ with } \bullet_1 \xrightarrow{+} \bullet_2 ::= \bullet_1 \to \bullet_2 | \bullet_1 \to \bullet \xrightarrow{+} \bullet_2$$

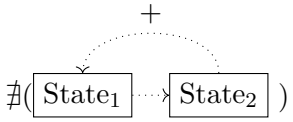
To apply HR* conditions to typed graphs the nodes and edges in its various graphs need to be typed and their morphisms must be typed graph morphisms similar to Definition 4. Hyperedges need no typing. Since the nodes connecting a hyperedge to the rest of the graph are already typed, it suffices to allow only properly typed terminal elements in the corresponding grammar. That is any terminal edge must be defined between nodes of allowed types as well. This ensures that any fully resolved grammar yields a proper typed graph.

Example 29. With this the missing requirements **R4** and **R5** of Section 4.3 can be encoded. In the following nodes are directly referred to as their type and dotted edges are the parent relation of the hierarchical finite state machine example.

There exists a top node that every other state has a parent path to:

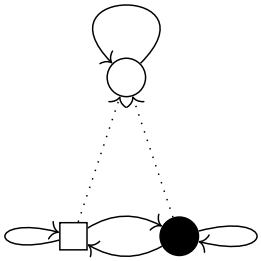


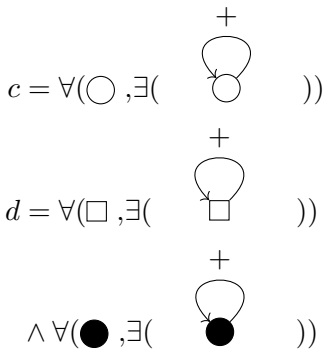
The parent relation has no non-trivial loops:



With the extension to typed graphs Theorem 2 can also be extended to these conditions. The construction has to also be applied to the replacement grammar, adding additional rules to each rule set where applicable.

Example 30. Consider the following constraint and meta model with its derived constraints:





For the replacement grammar this results in four different derived productions:

$$\circ_1 \xrightarrow{+} \circ_2 ::= \circ_1 \to \circ_2 | \circ_1 \to \circ \xrightarrow{+} \circ_2$$

$$\bullet_1 \xrightarrow{+} \bullet_2 ::= \bullet_1 \to \bullet_2 | \bullet_1 \to \bullet \xrightarrow{+} \bullet_2 | \bullet_1 \to \square \xrightarrow{+} \bullet_2$$

$$\square_1 \xrightarrow{+} \bullet_2 ::= \square_1 \to \bullet_2 | \square_1 \to \bullet \xrightarrow{+} \bullet_2 | \square_1 \to \square \xrightarrow{+} \bullet_2$$

$$\square_1 \xrightarrow{+} \square_2 ::= \square_1 \to \square_2 | \square_1 \to \bullet \xrightarrow{+} \square_2 | \square_1 \to \square \xrightarrow{+} \square_2$$

$$\bullet_1 \xrightarrow{+} \square_2 ::= \bullet_1 \to \square_2 | \bullet_1 \to \bullet \xrightarrow{+} \square_2 | \bullet_1 \to \square \xrightarrow{+} \square_2$$

With more expressive constraints the problem of checking model validity becomes more complex as for a given graph the corresponding HR-Grammar must be parsed. This may result in long run times although for certain grammars efficient parsing algorithms exist [DHM19].

List of Abbreviations

BPMN Business Process Model and Notation. 1

CDFG Control/Data Flow Graph. 28, 29

DPF Diagram Predicate Framework [Rut10]. 2, 3, 15, 17–19, 46, 48, 49, 53

EMF Eclipse Modeling Framework. 1

MR Model and Repair Framework. 21, 27, 30, 33, 46–48, 50, 53

SysML Systems Modeling Language. 1

UML Unified Modeling Language. 1

Bibliography

[ACD+15] P. Attie, A. Cherri, K. Dak Al Bab, M. Sakr, and J. Saklawi. Model and program repair via sat solving. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 148–157, Sep. 2015.

[BP16] Christopher Bak and Detlef Plump. Compiling graph programs to c. In Rachid Echahed and Mark Minas, editors, *Graph Transformation*, pages 102–117, Cham, 2016. Springer International Publishing.

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*, volume 138. Cambridge University Press, 2012.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

[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.

[DHM19] Frank Drewes, Berthold Hoffmann, and Mark Minas. Formalization and correctness of predictive shift-reduce parsers for graph grammars based on hyperedge replacement. *Journal of Logical and Algebraic Methods in Programming*, 104:303–341, 2019.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

[Fli16] Nils Erik Flick. *Proving Correctness of Graph Programs Relative to Recursively Nested Conditions*. PhD thesis, Universität Oldenburg, 2016.

[FvdA15] Dirk Fahland and Wil MP van der Aalst. Model repair—aligning process models to reality. *Information Systems*, 47:220–243, 2015.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and*

Computation Structures (FOSSACS 2001), LNCS 2030), pages 230–245, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[HS18] Annegret Habel and Christian Sandmann. Graph repair by graph programs. In *Graph Computation Models (GCM 2018), LNCS 11176*, pages 431–446, 2018.

[HS19] Annegret Habel and Christian Sandmann. Rule-based graph repair. In *Graph Computation Models (GCM 2019)*, pages 49–64, 2019.

[LNOP18] Leen Lambers, Marisa Navarro, Fernando Orejas, and Elvira Pino. Towards a navigational logic for graphical structures. In *Graph Transformation, Specifications, and Nets*, volume 10800, pages 124–141, 2018.

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

[NOPL16] Marisa Navarro, Fernando Orejas, Elvira Pino, and Leen Lambers. A logic of graph conditions extended with paths. In *Graph Computation Models (GCM 2016)*, 2016.

[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). LNCS 10374*, pages 171–181, 2017.

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

[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 *International Conference on Graph Transformation (ICGT 2014), LNCS 8571*, pages 33–48. Springer, 2014.

[Rad13] Hendrik Radke. HR* graph conditions between counting monadic second-order and second-order graph formulas. *Electronic Communications of the EASST*, 61, 2013.

[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.

[RMKL18] Fazle Rabbi, Lars Michael Kristensen, and Yngve Lamo. Static analysis of conformance preserving model transformation rules. pages 152–162, 01 2018.

[RRLW12] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in mde. *The Journal of Logic and Algebraic Programming*, 81(4):422–457, 2012.

[Rut10] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, 2010.

[SLO19] Sven Schneider, Leen Lambers, and Fernando Orejas. A logic-based incremental approach to graph repair. In *International Conference on Fundamental Approaches to Software Engineering*, pages 151–167. Springer, 2019.

[Ste07] Sandra Steinert. *The Graph Programming Language GP*. PhD thesis, University of York, 2007.

[TOLR17] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. Change-preserving model repair. In *Fundamental Approaches to Software Engineering (ETAPS 2017), LNCS 10202*, pages 283–299. Springer, 2017.

Erklärung

Hiermit versichere ich an Eides statt, 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.

Jens Sager
Matrikelnummer 4430077
Oldenburg, den 05. September 2019