Rule-Based Repair of EMF Models: Formalization and Correctness Proof

Nebras Nassar[1(✉)], Jens Kosiol[1], Hendrik Radke[2]

[1] Philipps-Universität Marburg, Germany,
{nassarn,kosiolje}@informatik.uni-marburg.de
[2] SAP SE, Germany,
hendrik.radke@sap.com

Abstract. In model-driven engineering (MDE), resolving model inconsistencies in a semi-interactive way promises to increase the productivity and quality of software development. In this paper, we discuss properties of an approach for model repair: The proposed algorithm can trim and complete models based on the Eclipse Modeling Framework (EMF), and thereby resolve their cardinality violations. We use rules, as defined in the theory of algebraic graph transformation, to declare the different actions executed during the particular steps of the algorithm. This formal background is used to reason for the correctness of the algorithm and to present conditions under which it always terminates. Possible adaptions of and more general use cases for the algorithm are discussed.

Keywords: Eclipse Modeling Framework (EMF) · Model Repair · Algebraic Graph Transformation · Correctness Proof

1 Introduction

Model-driven engineering (MDE) has become increasingly popular in various engineering disciplines, especially in software development. In MDE, models are the primary artifacts, whereas model transformations are the most important operations to change models. Domain-specific modeling languages promise to increase the productivity and quality of software developments. During the modeling process, a variety of model inconsistencies, e.g., cardinality violations, may occur for different reasons such as lack of information, misunderstandings, or the models' complexity. Thus, the modeling process has to be provided with tools, e.g., model repair in domain-specific model editors would raise their convenience.

There are a number of model repair approaches in the literature which, however, function either interactively such as [5,6,13,14] or automated, e.g., [1,9,10,19]. In [11] we present a rule-based approach to resolve cardinality violations in a semi-interactive way, i.e., the approach does not leave the resolution strategy completely to the modeler as in pure rule-based approaches. Instead, it semi-interactively guides modelers to yield valid models. The rule-based approach and the developed tool support are based on the Eclipse Modeling Framework (EMF) [8] and the model transformation language Henshin [2]. The tool can repair EMF models automatically or semi-interactively. Additionally, the approach is promising to be scalable.

In this paper, we formalize the approach presented in [11] using the theory of algebraic graph transformation and discuss its soundness: Two theorems state under which conditions the algorithm terminates and yields valid models. Two corollaries make further features of the algorithm explicit, e.g., reliability in the sense that the algorithm does not change valid models. We discuss possible adaptions of the algorithm and its usefulness in more general cases.

The rest of the paper is organized as follows: Section 2 introduces our running example. Section 3 presents the formal background of EMF-model graphs and transformations. In Sect. 4, we recall the rule-based algorithm for model repair. Section 5 proves the correctness of the rule-based algorithm and other features. In Sect. 6, we discuss the usefulness of the algorithm in more general settings. An overview on related work is given in Sect. 7, while Sect. 8 concludes the paper.

2 Running Example

In this section, we present the running example we will use to illustrate the algorithm and many of our definitions. Figure 1 shows the meta-model of a paper template. The root node is `Paper`. A `Paper` contains exactly one `Abstract` and `Bibliography`, at least one `Section` and maybe some `FloatObject`s, which are displayed in `Section`s. The class `FloatObject` is abstract and can either be instantiated by a `Table` or a `Figure`. A `Section` can contain `Section`s as `subsection`s and `Sentence`s. Up to three `HyperLink`s, which is an abstract class from which `Reference` and `Citation` inherit, are contained in a `Sentence`. They refer to `FloatObject`s or cite `BibItem`s contained in the `Bibliography`.

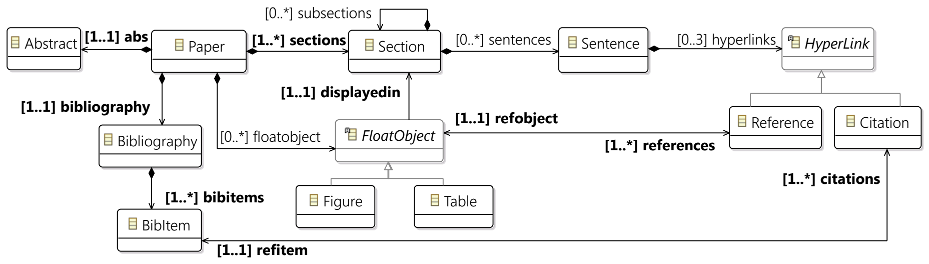


Fig. 1: Paper template meta-model

In Ex. 4 we present how the algorithm repairs an invalid instance shown in Fig. 2. The solid-line elements there represent the invalid instance. It consists of one root node `Paper` which contains a `Figure`-node and two `Bibliography`-nodes with one `BibItem`-node each. Several conditions of a valid instance are not met, e.g., an `Abstract`-node is missing, the `Figure`-node is not referenced and not displayed in any `Section`-node, and the number of `Bibliography`-nodes exceeds the upper bound, i.e., the upper bound of edge type `bibliography` is violated.

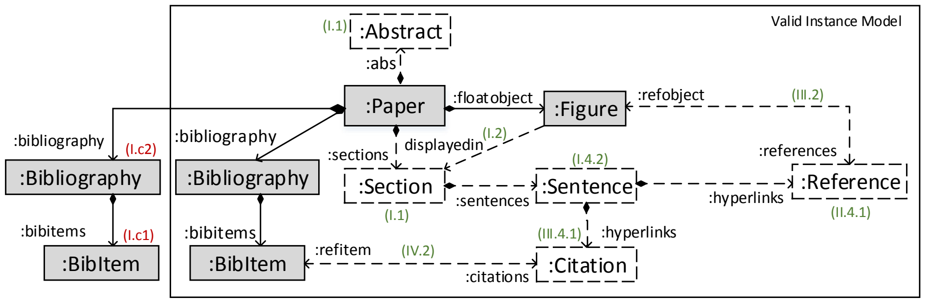


Fig. 2: The abstract syntax of the invalid Paper model being repaired

3 EMF Model Graphs and Consistent Transformations

Our goal is to repair instance models w.r.t. a given EMF model [20] using model transformations. A suitable formal framework to conduct this work is the theory of algebraic graph transformation [7]. When formalizing meta-modeling by this means, graphs occur at two levels: the type level (meta-models) and the instance level (models). This is reflected by the concept of *typed graphs*, where a fixed *graph TG* serves as an abstract representation of the type structure of a meta-model. Correct typing of instances is formalized by structure-preserving mappings of instance graphs to type graphs. Since setting values to empty required attributes is straightforward in EMF, we concentrate on multiplicity violations and EMF constraints in this paper. Thus, the theory of attributed typed graphs is not presented and the attributes are omitted in our example. This section recalls the background information needed to design valid EMF models and to change them in a consistency-preserving way. It is partially based on [4].

Definition 1. *A* graph $G = (V_G, E_G, s_G, t_G)$ *consists of a set* V_G *of* nodes *(or* vertices*), a set* E_G *of* edges*, and* source *and* target functions $s_G, t_G : E_G \to V_G$.

For graphs G *and* H*, a* graph morphism $f = (f_V, f_E)$ *is a pair of functions* $f_V : V_G \to V_H$ *and* $f_E : E_G \to E_H$ *such that* $f_V \circ s_G = s_H \circ f_E$ *and* $f_V \circ t_G = t_H \circ f_E$*. If* f_V *and* f_E *are inclusions, then* G *is a* subgraph *of* H*, short* $G \subseteq H$*.*

Definition 2. *A* type graph $TG = (T, I, A, C, OE, mult)$ *consists of a graph* $T = (V_T, E_T, s_T, t_T)$*, an* inheritance relation $I \subseteq V_T \times V_T$*, a set* $A \subseteq V_T$ *of* abstract nodes*, a set* $C \subseteq E_T$ *of* containment edges*, a relation* $OE \subseteq E_T \times E_T$ *of* opposite edges*, and a function* $mult : E_T \to \mathcal{N} \times (\mathcal{N} \cup \{*\})$ *subject to the following conditions: The inheritance relation* I *is a partial order and for each* $(i, j) \in mult(E_T)$ *it holds that* $i \leq j$ *and* $j \neq 0$ *(where* $\leq$ *is the standard order of natural numbers supplemented by* $i < *$ *for all* $i \in \mathcal{N}$*).*

We omitted the axioms that ensure that the relation OE models opposite edges.

The following concepts and notations will be used throughout the work and further clarify the correspondence between type graphs and EMF models: For $e \in OE$ we denote its opposite edge by e^{-1}. The *inheritance clan* of a node $n \in V_T$ is defined by $clan_I(n) = \{m \mid (m, n) \in I\}$. We write $m \leq n$ for

$m \in clan_I(n)$. We use the set of containment edges C to define a *containment relation* $cont_{TG} \subseteq V_T \times V_T$, which is given as the transitive closure of the relation $\{(n,m) \mid \exists c \in C : n \leq s(c) \wedge \; m \leq t(c)\}$ and write $n \sqsupseteq m$ for $(n,m) \in cont_{TG}$. In the following, by $NC := E_T \setminus C$ we refer to the non-containment edges. The function *mult* maps each edge of E_T to a *multiplicity*. A value of $*$ indicates an unconstrained number of edges. For $mult(e) = (i,j)$, i is called the *lower bound*, j is called the *upper bound* of e, and we define functions $low : E_T \to \mathcal{N}, upp : E_T \to \mathcal{N} \cup *$ which assign the respective bounds to each edge.

Example 1. The class diagram in Fig. 1 can be interpreted as type graph. For example `Section` $\sqsupseteq$ `HyperLink`, `Figure` $<$ `FloatObject`, $mult($`sentences`$) = (0,*)$, or $\texttt{refobject}^{-1} = \texttt{references}$.

Structure preserving maps establish the relationship between TG and instances:

Definition 3. *A graph G is called* typed over TG *if there exists a graph morphism $type_G = (type_{V_G}, type_{E_G}) : G \to TG$ s.t. $type_{V_G}(s_G(e)) \leq s_T(type_{E_G}(e))$ and $type_{V_G}(t_G(e)) \leq t_T(type_{E_G}(e))$ for all $e \in E_G$; $type_G$ is called* typing morphism. *A* typed graph morphism *between graphs G and H, which are both typed over TG, is a graph morphism $f = (f_V, f_E) : G \to H$ such that $type_{V_H} \circ f_V(n) \leq type_{V_G}(n)$ for all $n \in V_G$ and $type_{E_H} \circ f_E = type_{E_G}$.*

The typing of a graph G over TG with $type_G$ induces a containment relation on G. We write $C_G := \{e \in E_G \mid type_{E_G}(e) \in C\}$ and $cont_G$ is the transitive closure of the relation $\{(s_G(c), t_G(c)) \mid c \in C_G\}$. As above, we denote $(n,m) \in cont_G$ by writing $n \sqsupseteq m$. Analogously, we introduce the sets NC_G and OE_G.

To define typed graphs, which also respect the constraints of EMF and to later present our algorithm, we now give a list of certain properties typed graphs can possess and express these properties as logical formulas. Let G be a graph typed over TG by typing morphism $type_G$:

- **At-most-one-container:** No node of G has more than one container:
$$\forall c_1,c_2 \in C_G . t_G(c_1) = t_G(c_2) \Rightarrow c_1 = c_2 \quad .$$
- **No-containment-cycle:** No cycles of containment edges occur in G:
$$\forall n \in V_G . n \not\sqsupseteq n \quad .$$
- **No-parallel-edges:** No parallel edges exist, i.e., there are no two edges of the same type from the same source to the same target node:
$$\forall e,e' \in E_G \Big(\big(type_{E_G}(e) = type_{E_G}(e') \wedge s_G(e) = s_G(e') \wedge t_G(e) = t_G(e') \big) \Rightarrow e = e' \Big) \quad .$$
- **All-opposite-edges:** Each edge of opposite edge type has its inverse edge:
$$\forall e \in OE_G . \exists e' \in OE_G \Big(s_G(e) = t_G(e') \wedge t_G(e) = s_G(e') \wedge type_{E_G}(e) = type_{E_G}(e')^{-1} \Big) \quad .$$
- **Concreteness:** It instantiates no node of abstract type:
$$\forall n \in V_G . type_{V_G}(n) \notin A \quad .$$
- **Rootedness:** There exists a node r in V_G, called *root node*, such that all nodes of G, except for r, are transitively contained in r:
$$\forall n \in V_G \big(n \neq r \Rightarrow r \sqsupseteq n \big) \quad .$$

– **No-bound-violation:** For every node $n \in V_G$ and every edge type $e_T \in E_T$, for which n can serve as source node, the number of outgoing edges of type e_T from n is between e_T's lower and upper bound:

$$\forall n{\in}V_G.\forall e_T{\in}E_T\Big(type_{V_G}(n){\le}s_T(e_T){\Rightarrow}low(e_T){\le}\#\{e{\in}E_G \mid type_{E_G}(e){=}e_T{\wedge}s_G(e){=}n\}{\le}upp(e_T)\Big)\ .$$

This property, or partial satisfaction like **No-ub-violation**, can also be expressed using the following four ones:

- **No-nonCont-ub-violation:** For each node and each type of non-containment edges the number of outgoing edges of this type is smaller or equal to the upper bound of this edge type:

$$\forall n{\in}V_G.\forall e_T{\in}NC.\#\{e{\in}E_G \mid type_{E_G}(e){=}e_T{\wedge}s_G(e){=}n\}{\le}upp(e_T)\ .$$

- **Cont-ub-violation:** There exists a node with to many outgoing edges of a certain containment type:

$$\exists n{\in}V_G.\exists c{\in}C.\#\{e{\in}E_G \mid type_{E_G}(e){=}c \wedge s_G(e){=}n\}{>}upp(c)\ .$$

- **No-Cont-lb-violation:** Each node has (at least) the required number of outgoing containment edges:

$$\forall n{\in}V_G.\forall c{\in}C\Big(type_{V_G}(n){\le}s_T(c){\Rightarrow}\#\{e{\in}E_G \mid type_{E_G}(e){=}c \wedge s_G(e){=}n\}{\ge}low(c)\Big)\ .$$

- **nonCont-lb-violation:** For (at least) one node an outgoing edge of a certain non-containment type is missing:

$$\exists n{\in}V_G.\exists e_T{\in}NC\Big(type_{V_G}(n){\le}s_T(e_T)\ \wedge\ \#\{e{\in}E_G \mid type_{E_G}(e){=}e_T \wedge s_G(e){=}n\}{<}low(e_T)\Big)\ .$$

– **Missing-edge(e_T):** An edge of non-containment type e_T is missing:

$$\exists n{\in}V_G\Big(type_{V_G}(n){\le}s_T(e_T)\ \wedge\ \#\{e{\in}E_G \mid type_{E_G}(e){=}e_T \wedge s_G(e){=}n\}{<}low(e_T)\Big)\ .$$

– **No-target-exists(e_T):** No node of G can serve as target node for a non-containment edge of type e_T without violation of upper bounds:

$$\forall n{\in}V_G\Bigg(type_{V_G}(n){\not\le}t_T(e_T) \vee \Big(e_T{\in}OE \wedge \#\{e{\in}E_G \mid type_{E_G}(e){=}e_T \wedge t_G(e){=}n\}{=}upp(e_T^{-1})\Big)\Bigg)\ .$$

– **No-target-creation-possible(e_T):** For an edge of non-containment type e_T it is not possible to directly create a node, which can serve as target for an edge of this type:

$$\forall n{\in}V_G.\forall c_T{\in}C\Big(type_{V_G}(n){\le}s_T(c_T) \wedge t_T(e_T){\le}t_T(c_T)$$
$$\Rightarrow\ \#\{c{\in}C_G \mid type_{E_G}(c){=}c_T \wedge s_G(c){=}n\}{=}upp(c_T)\Big)\ .$$

For later use we additionally need to introduce two counters: For a given typed graph G the counter *ContUbViol(G)* gives the number of upper bound violations of containment type edges in G, which can be interpreted as the number of surplus nodes, while *nonContLbViol(G)* counts the number of lower bound violations of edges of non-containment type, i.e., the number of missing edges.

Different degrees of conformity to the EMF constraints are needed:

Definition 4. *A graph G typed over $TG = (T, I, A, C, OE, mult)$ with typing morphism $type_G$ is an* EMF-model graph *over TG (or w.r.t. TG) if the conditions* At-most-one-container*,* No-containment-cycle*,* No-parallel-edges*, and* All-opposite-edges *hold.*

When considering EMF-model graphs as instances of a meta-model, one expects them to be concrete and typically to be rooted. We did not include this in the above definition to be able to define rules more conveniently. But when repairing an instance, we suppose the input to be concretely typed and rooted and our aim is to receive such an EMF-model graph as output, since that is common practice in EMF. So, unless stated otherwise, when talking about EMF-model graphs in the following, we always assume them to be concrete and rooted; we use **EMFC** as a shortcut for this.

Definition 5. *A concrete and rooted EMF-model graph G is called* valid*, short* ***vEMFC****, if it additionally satisfies the* No-bound-violation *property.*

Usually, EMF models are expected to have at least one non-empty finite instance model or for each non-abstract class of the EMF model one instance which instantiates this class. We need to additionally introduce a stricter notion. We expect instances, which may violate lower, but no upper bounds, to be capable of being complemented into a valid EMF-model graph (without deletion of elements):

Definition 6. *An EMF model is called* finitely satisfiable *(f.s.) if there exists a non-empty finite instance. It is called* finitely instantiable *(f.i.) if for every non-abstract class there exists a finite EMF instance model, instantiating it. It is called* fully finitely instantiable *(f.f.i.) if it satisfies the property that for every given finite EMF-model graph G which respects upper, but may violate lower bounds there exists a finite EMF-model graph G', s.t. G is a subgraph of G'.*

By definition the following implications hold: *f.f.i.* $\Rightarrow$ *f.i.* $\Rightarrow$ *f.s.*

Example 2. (a) Fully finitely instantiable: The meta-model presented in Fig. 1 is f.f.i.: Since no containment cycle with all lower bounds ≥ 1 exists, the creation of missing nodes to fulfill all lower bounds of containment type edges will always result in a finite model. Thereafter, a `Section`-node will always exist so that all `FloatObject`-nodes can be displayed and the needed number of `Sentence`-nodes can be created inside, to ensure that all `FloatObject`-nodes can be referenced and all `BibItem`-nodes be cited. More generally, let TG be a type graph without cycles over containment edges with all lower bounds > 0 s.t. for every node $n \in (V_T \setminus A)$ the upper bound of at least one incoming containment relation is higher than the lower bound of all non-containment edges for which n can serve as target node. Then TG is f.f.i. This is particularly the case if all edges have unlimited upper bounds. *(b) Finitely instantiable:* Suppose the multiplicity of `reference` in Fig. 1 to be [1..1], s.t. every `FloatObject` is referenced by exactly one `HyperLink`, and the upper bound of `floatobject` to be limited, e.g. to 5. The result is a meta-model which is f.i., but not f.f.i.: One easily creates valid instances, but invalid instances with 6 or more nodes of type `Reference`, which do not violate any upper bound, cannot be complemented into a valid instance since maximally 5 nodes of type `FloatObject` can exist in a valid model and therefore the needed `references` can never be inserted. *(c) Not finitely satisfiable:* Suppose the containment type edge `subsection` in Fig. 1 to have multiplicity [1..$*$]. This is a typical example of a meta-model which is not f.s. or f.i.: The lower bound of 1 leads to an infinite chain of `subsections`.

When formalizing transformations of EMF models by the means of the theory of algebraic graph transformation, they are specified by transformation rules. Such rules consist of a left-hand side L and a right-hand side R specifying which elements are deleted or created. Elements in $K = L \cap R$ are preserved. In addition, positive (PACs) and negative (NACs) application conditions may ensure or forbid the existence of certain model patterns.

Definition 7. *A* transformation rule *over type graph TG is given by $p = (L \supseteq K \subseteq R,\ type,\ NAC,\ PAC)$, where L, K and R are EMF model graphs (not necessarily rooted), type is a triple of typing morphisms from L, K and R to TG, and NAC and PAC are sets of pairs $(N, type_N)$ with $L \subseteq N$ such that the following conditions hold: (1) Model elements are not retyped: $type_L \supseteq type_K \subseteq type_R$, (2) newly created nodes are concretely typed: $type_{V_R}(V_R \setminus V_K) \cap A = \emptyset$, and (3) types in NACs and PACs are always equal or finer than in L: for all $(N, type_N)$ in NAC or PAC and x in L, $type_N(x) \leq type_L(x)$.*

A *transformation step* $G \overset{p,m}{\Longrightarrow} H$ between two model graphs G and H is defined by first finding an injective typed morphism m, called *match*, of the left-hand side L of rule p to the current model graph G and then constructing H in two passes: (1) building $D := G \setminus m(L \setminus K)$, i.e., erasing all the graph items that are to be deleted, and (2) constructing $H := D \cup (R \setminus K)$, i.e., adding all the graph items that are to be created. The image of K under m determines the "location" of the creation. Moreover, a rule is only applicable with given match, if it is possible to extend the match to every PAC, but to no NAC. Note, m has to fulfill the *dangling condition*, i.e., all adjacent edges of a node to be deleted have to be deleted as well, such that D becomes a graph. A transformation step can also be formalized as double pushout, see [4, Def. 7].

One needs to pose further constraints on rules to ensure that they not only transform a concretely typed graph into a concretely typed graph, but an EMF-model graph into an EMF-model graph:

Example 3. Figure 4 shows an example for a rule in Henshin syntax: An edge of type `displayedin` gets inserted between two existing nodes of types `FloatObject` and `Section`, as long as no such edge already exists between those two nodes and the `FloatObject` is not already displayed. The design of the rule ensures two things: No parallel edge is created and the rule is only applicable as long as the lower bound of `displayedin` (here 1) is not already met. Note, that this also ensures that no upper bound violation (here also 1) is introduced, when applying this rule. The reason for choosing the lower and not the upper bound will become apparent, when we explain how we use this kind of rules in our algorithm in Sect. 4. Figure 3 gives a general design scheme for such rules in an intuitive notation: Given a non-containment edge type `ref` with a lower bound $m > 0$, the condition $NACp$ forbids to insert an edge from a source node of type `A` to a node of type `B` if there is already one of that type. Condition $NACm$ checks if the lower bound m has not yet been reached.

There are more EMF constraints to be respected, e.g., every newly created node has to be (transitively) contained in the root node. [4] give a set of conditions for

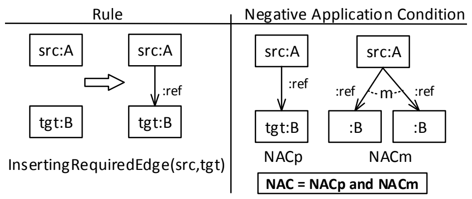


Fig. 3: Rule scheme for inserting a required non-containment edge

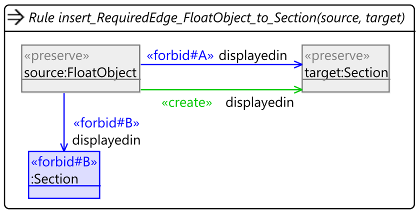


Fig. 4: An example rule for inserting a required *displayedin*-edge

such *consistent transformation rules* and prove it to be sufficient. We designed the rules we use to meet those conditions. I.e, applying any of these rules to an EMF-model graph results in an EMF-model graph again. All the different kinds of rules used are presented in [12], so our claim on their design can be checked.

4 A Rule-Based Approach of EMF Repair: An Overview

In this section, we first recall the rule-based algorithm for model repair presented in [11]. The algorithm consists of two main phases. Each phase is made up of several steps. The steps are declarative and use rules automatically derived from a given meta-model: For each step there exists a set of rules of a certain kind. In most steps, these rules are randomly applied as often as possible and the next step is called as soon as no rule is applicable anymore. More information about the algorithm and the design and derivation of the rules is found in [11,12].

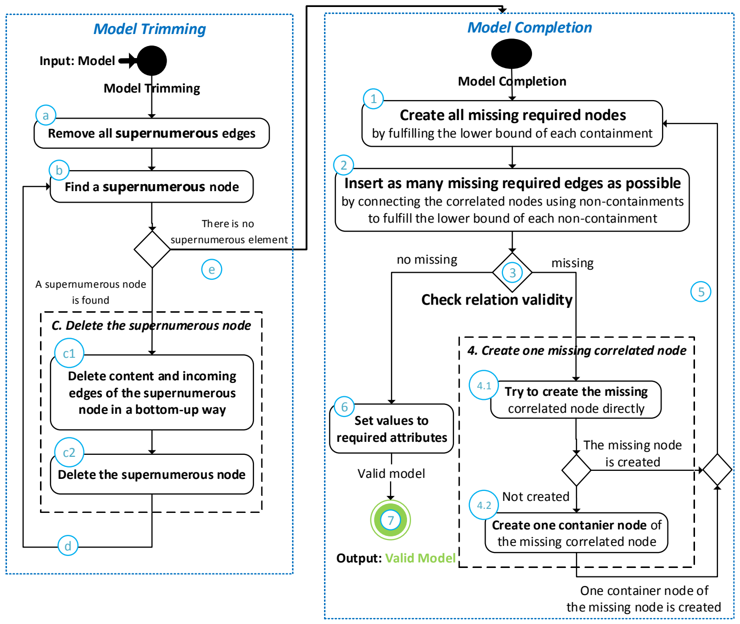
Above the description given there, we here introduce pre- and postconditions for each step. We will present those in a way reminding of Hoare triples: By $\langle\, PreX$, Step (X), $PostX\,\rangle$ we assert that, given a graph G satisfying $PreX$, the execution of Step (X) will lead to a graph G' satisfying $PostX$. In the following, we first present the algorithm's steps and afterwards the kinds of rules used in them; Fig. 5 shows the control flow of the algorithm.

Model trimming: This phase eliminates supernumerous model elements. It consists of the following steps: Step (a) removes *all supernumerous edges*, i.e., non-containment edges that exceed the upper bound of their respective type: $\langle\, EMFC$, Step (a), $EMFC \wedge$ *No-nonCont-ub-violation* $\rangle$.

Step (b) checks if there is *a supernumerous node*, i.e., a node which exceeds the upper bound of a containment edge: $\langle\, Post(a)$, Step (b), $Post(a)\,\rangle$.

If there is none, the model is ready to be completed (Step (e)). Otherwise, this node and its content have to be deleted. This is done in Step (c). It deletes all the incoming and outgoing edges of this node and its content nodes, and then deletes the content nodes in a bottom-up way (Step (c1)); thereafter, it deletes the node itself (Step (c2)): $\langle\, Post(a) \wedge$ *Cont-ub-violation*, Step (c), $Post(a) \wedge (ContUbViol(G) > ContUbViol(G'))\,\rangle$. Step (d) calls Step (b) again to check if there is another supernumerous node.

Model completion: This phase adds required model elements. It consists of the following steps: Step (1) creates *all missing required nodes*, i.e., nodes that are needed to fulfill the lower bound of a containment edge type: $\langle\, EMFC \wedge$



Model Trimming
Input: Model
Model Trimming
a
Remove all supernumerous edges
b
Find a supernumerous node
There is no supernumerous element
e
A supernumerous node is found
C. Delete the supernumerous node
c1
Delete content and incoming edges of the supernumerous node in a bottom-up way
c2
Delete the supernumerous node
d
Model Completion
Model Completion
1
Create all missing required nodes
by fulfilling the lower bound of each containment
2
Insert as many missing required edges as possible
by connecting the correlated nodes using non-containments to fulfill the lower bound of each non-containment
no missing
missing
3
Check relation validity
5
4. Create one missing correlated node
4.1
Try to create the missing correlated node directly
The missing node is created
6
Set values to required attributes
Valid model
7
Output: Valid Model
4.2
Not created
Create one contanier node of the missing correlated node
One container node of the missing node is created

Fig. 5: Model trimming and completion algorithm

No-ub-violation, Step (1), *EMFC* $\wedge$ *No-ub-violation* $\wedge$ *No-cont-lb-violation* $\rangle$. At the end of this stage, the model may contain nodes which are not connected by required non-containment edges.

Step (2) tries to insert *all missing required edges* by adding non-containment edges to the nodes in the model, in order to fulfill the lower bound of each non-containment edge type. This step may stop without having inserted all required edges due to potentially missing target nodes, i.e., it may happen that there is no further free node of a required type. Thus, we get $\langle$ *Post*(1), Step (2), *Post*(2) $\rangle$.
Post(2):= $\Big($*EMFC* $\wedge$ *no-bound-violation*$\Big)$ $\vee$ $\Big($*Post(1)* $\wedge$ $\exists$ e_T $\in$ *NC* $\big($*Missing-edge(e_T)* $\wedge$ *No-target-exists(e_T)*$\big)$ $\wedge$ *NonContLbViol(G)* $\geq$ *NonContLbViol(G')*$\Big)$.

Step (3) checks if a lower bound of a non-containment edge type is still violated. If all the required edges are already inserted, then we have a valid EMF-model graph. Otherwise, there is still at least one required edge missing. So, Step (3) checks which of the two possibilities of *Post*(2) is actually true and additionally returns the type e_T of one missing non-containment edge (if true), but does not change the instance.

If an edge was missing, Step (4) tries to create a target node for the missing edge type. Although there may be more than one target node missing, only one is added in Step (4). If it cannot be created directly (Step 4.1), a (transitive) container node of it is created (Step 4.2). So we receive the following pre- and postconditions for these steps, where e_T was returned by Step (3):

Pre(4.1):= *EMFC* $\wedge$ *No-ub-violation* $\wedge$ *No-cont-lb-violation* $\wedge$ *Missing-edge(e_T)* $\wedge$ *No-target-exists(e_T)*.

Post(4.1):= *EMFC* $\wedge$ *No-ub-violation* $\wedge$ *Missing-edge(e_T)* $\wedge$ $\big($*target-exists(e_T)* $\vee$ *(No-target-creation-possible(e_T)* $\wedge$ *No-cont-lb-violation)*$\big)$.

Pre(4.2):= EMFC $\wedge$ *No-ub-violation* $\wedge$ *No-cont-lb-violation* $\wedge$ *Missing-edge*(e_T) $\wedge$ *No-target-creation-possible*(e_T).

Post(4.2):= EMFC $\wedge$ *No-ub-violation* $\wedge$ *Missing-edge*(e_T) $\wedge$ $\exists n \in V_{G'}(n \notin V_G \wedge type_{V_{G'}}(n) \sqsupseteq t_T(e_T))$.

Note that the type of a missing target node may have several subtypes. In this case, there may be several possibilities to create a missing node, choosing one of these subtypes. A further variation point are node types with several containers. This non-determinism may be solved by user interaction or automatically by randomly picking one.

After Step (4), the next iteration of model completion starts by calling Step (1) again: Adding a node to the model may lead to new required nodes and edges. Starting a new iteration ensures that all those model elements are generated and that all missing nodes and edges will be created in the end. Once the model is a valid EMF model, i.e., EMF constraints and multiplicities are met, the values of all the empty required attributes are set (Step (6)).

Example 4 (Repairing the instance model). We illustrate our algorithm by applying it to the invalid instance shown in Fig. 2. Note, the annotations at the model elements there refer to the corresponding algorithm pass and step numbers, whereas the dashed elements are elements created during the repair process. During the model trimming phase, no supernumerous edge has to be removed (Step (a)); Step (b) returns one `Bibliography` as supernumerous node. Its content gets deleted (c1) and then the `Bibliography` itself (c2). The next call of Step (b) does not return further supernumerous nodes and the second phase of the algorithm starts: In Step (1) of the first pass, an `Abstract`-node and a `Section`-node get created, since their respective incoming containment edges have lower bounds greater 0. The required `displayedin`-edge is inserted in Step (2) since a `Section`-node now exists. Step (3) then either returns an edge type `references` or an edge type `citations` as missing; we assume the first one to be the case. Since it is not possible to add a node of type `Reference` directly, a container (`Sentence`-node) for it is added in Step (4.2). In Step (4.1) in the second pass of the algorithm, the needed node of type `Reference` now is created. In the third pass it is possible to insert the edge of type `references`; in Step (3) of the third pass, an edge of type `citations` is recognized as missing and since a node of type `Sentence` now exists, the missing node of type `Citation` is added in Step (4.1). In the fourth pass, the last missing edge (`citations`) is inserted in Step (2). No further missing edges are recognized in Step (3) and thus the algorithm stops.

The Kinds of Rules: In Table 1 we give an overview over the kinds of rules we use in the different steps of the algorithm. We presented the design-scheme of *Required-edge-insertion* rules in Fig. 3: They create all possible, but only required edges, because *NACm* respects the lower bound. The other kinds of rules are designed in a similar fashion. A complete overview with example for every kind of rule is given in [12].

Table 1: Kinds of rules

Kind of the rule	Description
Required-node-creation	Create a node with its containment edge if the lower bound of its containment type is not yet reached (Step (1))
Required-edge-insertion	Insert a non-containment edge if the lower bound of the corresponding non-containment type is not yet reached and there is no parallel edge (Step (2))
Required-edge-checking	Check if a required non-containment edge is missing (Step (3))
Additional-node-creation	Add a node with its containment edge if the upper bound of its containment type is not yet reached (Step (4))
Exceeding-edge-removing	Remove a non-containment edge if the upper bound of the corresponding non-containment type is exceeded (Step (a))
Exceeding-node-finding	Find a node that exceeds the upper bound of the corresponding containment type (Step (b))
Node-content-deleting	Delete a (transitively) contained leaf node with its containment edge from a given node (Step (c))
Element-deleting	Delete a given element (i.e., a non-containment edge or a leaf node with its containment edge) directly (Step (c))

5 Correctness Proof of the Algorithm

In this section, we present and proof some properties of the algorithm. During the proofs we will not argue that the condition *EMFC* stays satisfied since that is true by construction of the rules (compare the end of Sect. 3). In the following, let TG be a fixed type graph over which all the occuring graphs are typed.

Our trimming of models restores integrity with respect to upper bounds:

Theorem 1. *The* model trimming *algorithm in Sect. 4 is correct, i.e., given an EMF-model graph G, the algorithm terminates in a finite number of steps, yielding an EMF-model graph G' that satisfies all upper bounds of TG.*

Proof. For a graph G, let $ub(G)$ denote the sum of the number of upper bound violations of (1) containment and (2) non-containment relations. Let G' denote a graph resulting from the application of one of the rules of kinds that constitute the Steps (a) – (c) of the algorithm (compare Table 1) on graph G. If all upper bounds of G are satisfied, $ub(G) = 0$, Step (a) and (b) do not change the graph, and the trimming algorithm terminates.

Otherwise, an upper bound is unsatisfied. Step (a) removes all supernumerous edges that violate the upper bounds of a non-containment relation, so $ub(G') \leq ub(G)$ and *No-nonCont-ub-violation* is certainly true afterwards. Step (b) checks for supernumerous nodes that violate the upper bounds of a containment relation, without changing the graph, so $ub(G) = ub(G')$ and the postcondition of this step equals its precondition. Step (c1) does two things to a supernumerous node v: (1) If v is a container, all content nodes of v (and their edges) are deleted in a bottom-up way. This may remove, but not add supernumerous nodes, so $contUbViol(G') \leq contUbViol(G)$ and equally $ub(G') \leq ub(G)$. If v is not a container, the graph remains unchanged. (2) Incoming edges of v are removed. In Step (c2), node v itself is removed, so $ub(G') < ub(G)$ and equally $contUbViol(G') < contUbViol(G)$.

Because $ub(G)$ decreases with every iteration, the algorithm terminates with a graph which does not violate any upper bound as result. $\square$

For the completion phase of our algorithm to work, we need the meta-model to exhibit certain qualities: It needs to be always possible to create missing target nodes for required edges (guaranteed by full finite instantiability) and the creation of one (of the) possible target node(s) for a required edge may not lead to the need to create a node of the same type again infinitely often. We characterize precisely how to prevent this second problem:

Sometimes, when a node has to be created to serve as target for a required non-containment edge, there exist essentially different ways to do that. This can be due to two causes: There exist different containment paths from the root to the target node in the sense that in one path a node type occurs that does not occur in another path. Or because of inheritance there are different possible node types that can serve as target node. We need to pay closer attention to situations like these if the required edge is an opposite edge.

Definition 8. *An opposite edge $e \in OE$ is called* critical *if $low(e) \geq 1$ and there either exists more than one possible target node type, i.e., $\#(clan_I(t_T(e)) \cap (V_T \setminus A)) > 1$, or there are containment paths p, p' for a target node type for e s.t. a node type occurs in one path that does not occur in the other.*

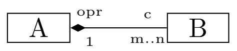
If e is such a critical edge, we require one of the following two conditions to hold: (a) The opposite edge e^{-1} has unlimited upper bound, or (b) (one of) the target node type(s) has at least one incoming containment edge with unlimited upper bound and if there are different possible target node types, they have to be able to serve as sources of exactly the same edge types:

Definition 9. *A type graph TG* satisfies the critical edge check condition (CECC) *if every critical edge e satisfies condition (a) or condition (b), presented below:* Condition (a) *on e demands $upp(e^{-1}) = *$. Condition (b) on e demands (b1) that there exists a containment edge $c \in C$ with $t_T(e) \leq t_T(c)$ and $upp(c) = *$. And if $clan_I(t_T(e)) \neq \{t_T(e)\}$ it additionally demands (b2) that for each two nodes $n, n' \in clan_I(t_T(e))$ and every edge $e' \in E_T$*

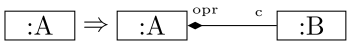
$$n \leq s_T(e') \Leftrightarrow n' \leq s_T(e') \ . \tag{1}$$

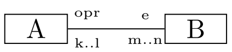
Theorem 2. *Let TG be a fully finitely instantiable type graph that meets the critical edge check condition. Then the* model completion *from Sect. 4 is correct, i.e., given an EMF-model graph G, which satisfies* No-ub-violation*, the algorithm terminates after a finite number of steps, yielding a valid EMF-model graph G'.*

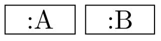
Proof. Let TG be a type graph and G a graph typed by TG s.t. the conditions formulated above hold. We check termination and the satisfaction of the postconditions formulated in Sect. 4 for each step and then argue for overall termination.

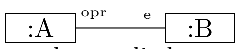
Step (1). For every containment edge type $c =$ 

in TG,

the set of *required-node-creation* rules contains a rule  with a negative application condition *ac* that prevents the rule from being applied at nodes that already meet the lower bound m. But as long as it is not met, the rule is applicable. Since $m \neq *$ by definition, those rules can only be applied finitely often with the same match. Therefore the only possibility for Step 1 to not terminate is a containment-cycle with all lower bounds > 0, which is excluded since TG is f.f.i. (f.i. would even be enough). Thus, Step (1) terminates with a graph G' as result where *No-cont-lb-violation* holds.

Step (2). For every non-containment edge type $e =$ 

in TG, the set of *required-edge-insertion* rules contains a rule 

$\Rightarrow$ 

with an application condition *ac* ensuring that the rule can not be applied to nodes of types A and B if there is already an edge of type e between them or the lower bound of A or the upper bound of B (if it is an edge of opposite type) are already fulfilled. Because of these application conditions and since no new nodes are created during the execution, Step (2) terminates after finitely many steps as soon as every lower bound of non-containment relations is satisfied, or there is no suitable target node available for any unsatisfied non-containment relation (this is exactly the statement of *Post(2)*). Note, that the result G' of this step of the algorithm is independent of the order of execution and choice of matches in the following sense: The number of missing non-containment edges of a certain type is equal for every possible resulting graph. Only the places, where edges are missing, can be different.

Step (4). If a *required-edge-checking* rule was applicable in Step (3), a target node for a certain non-containment edge type is missing. Since we assume TG to be f.f.i., at least one possible target node can be created without violating any upper bounds. If no target node can be created directly, at least a possible container can. These two possibilities are reflected by *Post(4.1)*, *Pre(4.2)*, and *Post(4.2)*. As soon as such a node is created, Step (4) is finished and Step (5) calls Step (1) again.

Steps (3) and (6). *Required-edge checking* rules do not change the graph and only finitely many checks are performed, so Step (3) always terminates. Also, the setting of required empty attributes is finished after finitely many steps and does not change anything which influences the kind of validity we are discussing.

Overall termination. Let e be a critical edge and let $low(e) = m$ denote its lower bound. We assume TG to meet the (CECC). Suppose that e satisfies condition (a) of Def. 9. This implies $upp(e^{-1}) = *$. If it was not possible to create an edge of type e during Step (2) often enough, we have at some point created m appropriate target nodes for an edge of type e. Since each of those target nodes can serve as target for an unlimited number of edges of type e, after this point all edges of type e will be created during Step (2). Now suppose that e satisfies condition (b) of Def. 9. Then it is ensured that at some point a target node for e can always be created directly, because at least one possible container of a target node has an unlimited upper bound: If a target node for e had to be created often enough, the randomness of the creation of target nodes (if no direct container is available) makes for that. After that point, it is always possible to create the target node directly. Additionally, if more than one possible target node exists, Eq. 1 states that it makes no difference for the newly arising required containment and non-containment edges, which target is created. In summary, after finitely many iterations of the algorithm, the algorithm becomes deterministic: Every time Step (3) is called, the only missing edges are of types for which there exists only one way to create a target node, or for which it makes no difference which target node is created. Since we assume TG to be f.f.i., and we only create elements that need to exist in a valid model, the algorithm altogether terminates. The resulting graph needs to be a valid EMF-model graph since this is the only situation in which the algorithm stops. □

The above proofs give us almost immediately two further results:

Corollary 1. *If the algorithm terminates, when applied to an EMF-model graph G, the result is a valid EMF-model graph G'.*

Proof. The proofs of the previous theorems showed that we apply rules as long as there exist violations of upper or lower bounds. Thus there are only three possible outcomes when starting the algorithm: No termination, break of the algorithm in Step (4), because it is not possible to create a needed target node, or termination with a valid model. □

Corollary 2. *If G already is a valid EMF-model graph, application of the algorithm to G will result in G again.*

Proof. By construction of the rules, as discussed in the proofs of Theorems 1 and 2, the applicability of rules that create or delete graph elements of valid models is prevented by suitable designed application conditions. □

6 Discussion

In this section, we discuss the usefulness of the approach in more general settings.

Moving Instead of Trimming. In some cases moving the elements into other containers (sources) without violating their upper bounds instead of deleting them would be possible. But this does not always work: E.g., the consistency requirements for rules which move cycle-capable containment edges are quite strict [4]. And more importantly because of finite upper bounds it is possible that there is a maximal number of instances of a certain node type which are allowed to occur in EMF-model graphs. No graph that contains more nodes for one type n than allowed can be repaired without deletion. But as long as this is considered, the moving of supernumerous elements into other already existing containers (sources) or newly created ones (if possible) does not change the validity of Thm. 1 and therefore neither that of Thm. 2.

Finitely Instantiable Versus Fully Finitely Instantiable Meta-Models. While deleting during the trimming part of the algorithm, we intended not to delete during the second part at the cost of not being able to repair every instance of finitely instantiable type graphs TG: Deletion would have to take place in Step (4) of the algorithm if no target node can be created. In this case, the bounds of all containment edges would be satisfied, but the lower bound of at least one non-containment edge violated. This means that multiplicities of non-containment edges would not only demand or forbid the existence of non-containment edges, but additionally impose constraints on the number of certain types of nodes. Here, it may be more advisable to adapt the multiplicities of the meta-model than to loose information by deletion.

To summarize, when applying the considered approach to finitely, but not fully finitely instantiable meta-models, the following possibilities exist: (a) Instance gets repaired. (b) Algorithm breaks in Step 4 because a needed target node cannot be created. A solution might be achieved by creating finitely many new nodes of root type, or there is no termination, even when creating new nodes of root type. (c) No termination. Characterizing these situations by features of the type graph or the instance which is to be repaired is a future work.

7 Related Work

We briefly relate our work to rule-based approaches and then to logic-based approaches. More detailed information can be found at [11].

Rule-based approaches such as [5,6,13,14,16] present rule-based techniques for fixing inconsistencies at different locations in models and predicting side effects. A novel evaluation is carried out to show that the (translated) choices for fixing inconsistencies in a single location are correct and complete. In all these approaches, the resulting models are not shown to be valid. Additionally, the user has to find manually a way to repair the whole model (if any).

Several logic-based approaches such as [1,9,10,18,19] provide support for automatic inconsistency resolution from a logical perspective. Partial models, their meta-models, and additional constraints are translated into logical formulas. Solvers such as SAT solver (Alloy analyzer) or constraint logic program are used to satisfy them, thereafter. The translation is evaluated using several experiments or test cases. All these approaches provide automatic model repair. In [15], the authors developed a search-based tool which uses a regression planning algorithm in Prolog to find and resolve inconsistencies through the generation of one or more resolution plans. The approach is evaluated through several stress-tests. These approaches do not allow user interactions during model repair.

8 Conclusion

In this work, we proved the correctness of an algorithm that can repair EMF models automatically or semi-automatically and discussed its usability. Using the theory of algebraic graph transformation as formal background, we were able to give precise conditions under which the algorithm is guaranteed to terminate. Furthermore, we showed its reliability: The resulting models are valid and conform to all EMF constraints and thus can be opened by the EMF editor. Since it is possible to translate (Essential) OCL constraints into graph constraints [3,17], we are confident that at least an important subset of them could be dealt with in a similar way and we plan to expand our approach in this direction.

Acknowledgement We are grateful to Gabriele Taentzer and the anonymous reviewers for their helpful comments. This work was funded by the German Research Foundation (DFG), projects "Generating Development Environments for Modeling Languages" and "Triple Graph Grammars (TGG) 2.0".

References

1. Apt, K.R., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge Univ. Press, Leiden (2006)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: MODELS, pp. 121–135. Springer (2010)
3. Bergmann, G.: Translating OCL to Graph Patterns. In: MoDELS, pp. 670–686. Springer (2014)
4. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. (SoSyM) pp. 227–250 (2012)
5. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: ICSE (2007)
6. Egyed, A., Letier, E., Finkelstein, A.: Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In: IEEE/ACM. pp. 99–108 (2008)
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS, Springer (2006)
8. Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf
9. Hegedüs, Á., Horváth, Á., Ráth, I., Branco, M.C., Varró, D.: Quick fix generation for DSMLs. In: VL/HCC. pp. 17–24. IEEE (2011)
10. Macedo, N., Guimarães, T., Cunha, A.: Model repair and transformation with Echo. In: ASE. pp. 694–697. IEEE (2013)
11. Nassar, N., Radke, H., Arendt, T.: Rule-based Repair of EMF Models: An Automated Interactive Approach. In: In Proc. ICMT. Springer (2017)
12. EMF Model Repair. http://uni-marburg.de/Kkwsr
13. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: A Consistency Checking and Smart Link Generation Service. ACM 2(2), 151–185 (2002)
14. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: Software Engineering. pp. 455–464. IEEE (2003)
15. Puissant, J.P., Straeten, R.V.D., Mens, T.: Resolving model inconsistencies using automated regression planning. SoSyM pp. 461–481 (2015)
16. Rabbi, F., Lamo, Y., Yu, I.C., Kristensen, L.M., Michael, L.: A Diagrammatic Approach to Model Completion. In: (AMT)@ MODELS (2015)
17. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations. In: Graph Transformation, pp. 155–170. Springer (2015)
18. Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A Methodology for Verifying Refinements of Partial Models. Journal of Object Technology pp. 3:1–31 (2015)
19. Sen, S., Baudry, B., Precup, D.: Partial Model Completion in Model Driven Engineering using Constraint Logic Programming. In: In Proc. INAP'07 (2007)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)