Graph Consistency as a Graduated Property

Consistency-Sustaining and -Improving Graph Transformations

Jens Kosiol[1] ✉, Daniel Strüber[2], Gabriele Taentzer[1], and Steffen Zschaler[3]

[1] Philipps-Universität Marburg, Marburg, Germany
{kosiolje,taentzer}@mathematik.uni-marburg.de
[2] Radboud University, Nijmegen, the Netherlands d.strueber@cs.ru.nl
[3] King's College London, London, UK szschaler@acm.org

Abstract. Where graphs are used for modelling and specifying systems, consistency is an important concern. To be a valid model of a system, the graph structure must satisfy a number of constraints. To date, consistency has primarily been viewed as a binary property: a graph either is or is not consistent with respect to a set of graph constraints. This has enabled the definition of notions such as constraint-preserving and constraint-guaranteeing graph transformations. Many practical applications—for example model repair or evolutionary search—implicitly assume a more graduated notion of consistency, but without an explicit formalisation only limited analysis of these applications is possible. In this paper, we introduce an explicit notion of consistency as a graduated property, depending on the number of constraint violations in a graph. We present two new characterisations of transformations (and transformation rules) enabling reasoning about the gradual introduction of consistency: while consistency-sustaining transformations do not decrease the consistency level, consistency-improving transformations strictly reduce the number of constraint violations. We show how these new definitions refine the existing concepts of constraint-preserving and constraint-guaranteeing transformations. To support a static analysis based on our characterisations, we present criteria for deciding which form of consistency ensuring transformations is induced by the application of a transformation rule. We illustrate our contributions in the context of an example from search-based model engineering.

Keywords: Graph Consistency · Graph Transformation Systems · Evolutionary Search · Graph Repair

1 Introduction

Graphs and graph transformations [8] are a good means for system modelling and specification. Graph structures naturally relate to the structures typically found in many (computer) systems and graph transformations provide intuitive

tools to specify the semantics of a model or implement refinement and analysis techniques for specifications.

In all of these scenarios, it is important that the graphs used are consistent; that is, that their structures satisfy a set of constraints. Some constraints can be captured by typing graphs over so-called type graphs [8]—these allow capturing basic structural constraints such as which kinds of nodes may be connected to each other. To allow the expression of further constraints, the theory of nested graph constraints has been introduced [11]. A graph is considered consistent if it is correctly typed and satisfies all given constraints. Note that this notion of consistency is binary: a graph either is consistent or it is not consistent. It is impossible to distinguish different degrees of consistency.

In software engineering practice, it is often necessary to live with, and manage, a degree of inconsistency [23]. This requires tools and techniques for identifying, measuring, and correcting inconsistencies. In the field of graph-based specifications, this has led to many practical applications, where a more fine-grained notion of graph consistency is implicitly applied. For example, research in model repair has aimed to automatically produce graph-transformation rules that will gradually improve the consistency of a given graph. Such a rule may not make a graph completely consistent in one transformation step, but performing a sequence of such transformations will eventually produce a consistent graph (*e.g.,* [12,21,22,25]). In the area of search-based model engineering (*e.g.,* [5,9]), rules are required to be applicable to inconsistent graphs and, at least, not to produce new inconsistencies. In earlier work, we have shown how such rules can be generated at least with regard to multiplicity constraints [5]. However, in all of these works, the notion of "partial" graph consistency remains implicit. Without explicitly formalising this notion, it becomes difficult to reason about the validity of the rules generated or the correctness of the algorithm by which these rules were produced.

In this paper, we introduce a new notion of graph consistency as a graduated property. A graph can be consistent *to a degree,* depending on the number of constraint violations that occur in the graph. This conceptualisation allows us to introduce two new characterisations of graph transformations: a *consistency-sustaining* transformation does not decrease the overall consistency level, while a *consistency-improving* transformation strictly decreases the number of violations in a graph. We lift these characterisations to the level of graph transformation rules, allowing rules to be characterised as consistency sustaining and consistency improving, respectively. We show how these definitions fit with the already established terminology of constraint-preserving and constraint-guaranteeing transformations / rules. Finally, we introduce formal criteria that allow checking whether a given graph-transformation rule is consistency sustaining or consistency improving w.r.t. constraints in specific forms.

Thus, the contributions of our paper are:

1. We present the first formalisation of graph consistency as a graduated property of graphs;

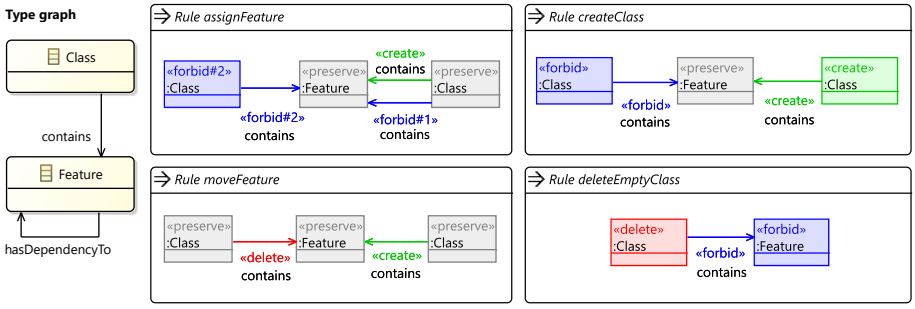


Fig. 1. Type graph and four mutation rules for the CRA problem.

2. We present two novel characterisations of graph transformations and transformation rules with regard to this new definition of graph consistency and show how these refine the existing terminology;
3. We present static analysis techniques for checking whether a graph-transformation rule is consistency sustaining or improving.

The remainder of this paper is structured as follows: We introduce a running example in Sect. 2 before outlining some foundation terminology in Sect. 3. Section 4 introduces our new concepts and Sect. 5 discusses how graph-transformation rules can be statically analysed for these properties. A discussion of related work in Sect. 6 concludes the paper. All proofs are provided in an extended version of this paper [16].

2 Example

Consider *class responsibility assignment* (CRA, [4]), a standard problem in object-oriented software analysis. Given is a set of features (methods, fields) with dependencies between them. The goal is to create a set of classes and assign the features to classes so that a certain *fitness function* is maximized. The fitness function rewards the assignment of dependent features to the same class (cohesion), while punishing dependencies that run between classes (coupling) and solutions with too few classes. Solutions can be expressed as instances of the type graph shown in the left of Fig. 1. For realistic problem instances, an exhaustive enumeration of all solutions to find the optimal one is not feasible.

Recently, a number of works have addressed the CRA problem via a combination of graph transformation and meta-heuristic search techniques, specifically evolutionary algorithms [10,28,5]. An evolutionary algorithm uses genetic operators such as cross-over and mutation to find optimal solution candidates in an efficient way. In this paper, we focus on mutation operators, which have been specified using graph transformation rules in these works.

Figure 1 depicts four mutation rules for the CRA problem, taken from the available MDEOptimiser solution [6]. The rules are specified as graph transformation rules [8] in the Henshin notation [1,29]: Rule elements are tagged as

delete, *create*, *preserve* or *forbid*, which denotes them as being included in the LHS, the RHS, in both rule sides, or a NAC. Rule *assignFeature* assigns a randomly selected as-yet-unassigned feature to a class. Rule *createClass* creates a class and assigns an as-yet-unassigned feature to it. Rule *moveFeature* moves a feature between two classes. Rule *deleteEmptyClass* deletes a class to which no feature is assigned.

Solutions in an optimization problem such as the given one usually need to be consistent with regard to the constraints given by the problem domain. We consider three constraints for the CRA case:

(c_1) Every feature is contained in at most one class.
(c_2) Every class contains at least one feature.
(c_3) If a feature F_1 has a dependency to another feature F_2,
and F_2 is contained in a different class than F_1,
then F_1 must have a dependency to a feature F_3 in the same class.

Constraints c_1 and c_2 come from Fleck et al.'s formulation of the CRA problem [10]. Constraint c_3 can be considered a *helper constraint* (compare *helper objectives* [13]) that aims to enhance the efficiency of the search by formulating a constraint with a positive impact to the fitness function: Assigning dependent features to the same class is likely to improve coherence.

Given an arbitrary solution model (valid or invalid), mutations may introduce new violations. For example, applying *moveFeature* can leave behind an empty class, thus violating c_2. While constraint violations can potentially be removed using repair techniques [22,12,25], these can be computationally expensive and may involve strategies that lead to certain regions of the search space being preferred, threatening the efficiency of the search. Instead, it would be desirable to design mutation operators that impact consistency in a positive or at least neutral way. Each application of a mutation rule should contribute to some particular violations being removed, or at least ensure that the degree of consistency does not decrease. Currently, there exists no formal framework for identifying such rules. The established notions of constraint-preserving and constraint-guaranteeing rules [11] assume an already-valid model or a transformation that removes all violations at once; both are infeasible in our scenario.

3 Preliminaries

Our new contributions are based on typed graph transformation systems following the double-pushout approach [8]. We implicitly assume that all graphs, also the ones occurring in rules and constraints, are typed over a common type graph TG; that is, there is a class $Graph_{TG}$ of graphs typed over TG. A *nested graph constraint* [11] is a tree of injective graph morphisms.

Definition 1 ((Nested) graph conditions and constraints). *Given a graph P, a* (nested) graph condition *over P is defined recursively as follows:* `true` *is a graph condition over P and if $a : P \hookrightarrow C$ is an injective morphism and d is*

a graph condition over C, $\exists\,(a : P \hookrightarrow C, d)$ is a graph condition over P again. If d_1 and d_2 are graph conditions over P, $\neg d_1$ and $d_1 \wedge d_2$ are graph conditions over P. A (nested) graph constraint *is a condition over the empty graph $\emptyset$.*

A condition or constraint is called linear *if the symbol $\wedge$ does not occur, i.e., if it is a (possibly empty) chain of morphisms. The* nesting level *nl of a condition c is recursively defined by setting $nl(\texttt{true}) \coloneqq 0$, $nl(\exists\,(a : P \hookrightarrow C, d)) \coloneqq nl(d)+1$, $nl(\neg d) \coloneqq nl(d)$, and $nl(d_1 \wedge d_2) \coloneqq \max(nl(d_1), nl(d_2))$. Given a graph condition c over P, an injective morphism $p : P \hookrightarrow G$* satisfies *c, written $p \models c$, if the following applies: Every morphism satisfies $\texttt{true}$. The morphism p satisfies a condition of the form $c = \exists\,(a : P \hookrightarrow C, d)$ if there exists an injective morphism $q : C \hookrightarrow G$ such that $p = q \circ a$ and q satisfies d. For Boolean operators, satisfaction is defined as usual. A graph G satisfies a graph constraint c, denoted as $G \models c$, if the empty morphism to G does so. A graph constraint c_1* implies *a graph constraint c_2, denoted as $c_1 \Rightarrow c_2$, if $G \models c_1 \Rightarrow G \models c_2$ for all graphs G. The constraints are* equivalent*, denoted as $c_1 \equiv c_2$, if $c_1 \Rightarrow c_2$ and $c_2 \Rightarrow c_1$.*

In the notation of graph constraints, we drop the domains of the involved morphisms and occurrences of `true` whenever they can unambiguously be inferred. For example, we write $\exists(C, \neg\exists C')$ instead of $\exists(\emptyset \hookrightarrow C, \neg\exists(a : C \hookrightarrow C', \texttt{true}))$. Moreover, we introduce $\forall(C, d)$ as an abbreviation for the graph constraint $\neg\exists(C, \neg d)$. Further sentential connectives like $\vee$ or $\Rightarrow$ can be introduced as abbreviations as usual (which is irrelevant for linear constraints).

We define a normal form for graph conditions that requires that the occurring quantifiers alternate. For every linear condition there is an equivalent condition in this normal form [25, Fact 2].

Definition 2 (Alternating quantifier normal form (ANF)). *A linear condition c with $nl(c) \geq 1$ is in* alternating quantifier normal form *(ANF) when the occurring quantifiers alternate, i.e., if c is of the form $Q(a_1, \bar{Q}(a_2, Q(a_3, \dots)\dots)$ with $Q \in \{\exists, \forall\}$ and $\bar{\exists} = \forall, \bar{\forall} = \exists$, none of the occurring morphisms a_i is an isomorphism, and the only negation, if any, occurs at the innermost nesting level (i.e., the constraint is allowed to end with $\texttt{false}$). If a constraint in ANF starts with $\exists$, it is called* existential*, otherwise it is called* universal*.*

Lemma 1 (Non-equivalence of constraints in ANF). *Let $c_1 = \exists(C_1, d_1)$ and $c_2 = \forall(C_2, d_2)$ be constraints in ANF. Then $c_1 \not\equiv c_2$.*

We have $c_1 \not\equiv c_2$ since $\emptyset \models c_2$ but $\emptyset \not\models c_1$. Lemma 1 implies that the first quantifier occurring in the ANF of a constraint separates linear constraints into two disjoint classes. This ensures that our definitions in Section 4 are meaningful.

Graph transformation is the rule-based modification of graphs. The following definition recalls graph transformation as a double-pushout.

Definition 3 (Rule and transformation). *A* plain rule *r is defined by $p = (L \hookleftarrow K \hookrightarrow R)$ with L, K, and R being graphs connected by two graph inclusions. An application condition ac for p is a condition over L. A* rule *$r = (p, ac)$ consists of a plain rule p and an application condition ac over L.*

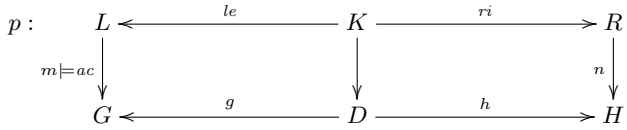


Fig. 2. Rule application

A transformation (step) $G \Rightarrow_{r,m} H$ *which applies rule* r *to a graph* G *consists of two pushouts as depicted in Fig. 2. Rule* r *is* applicable *at the injective morphism* $m : L \to G$ *called* match *if* $m \models ac$ *and there exists a graph* D *such that the left square is a pushout. Morphism* n *is called* co-match. *Morphisms* g *and* h *are called* transformation morphisms. *The* track morphism *[24] of a transformation step* $G \Rightarrow_{r,m} H$ *is the partial morphism* $tr : G \dashrightarrow H$ *defined by* $tr(x) = h(g^{-1}(x))$ *for* $x \in g(D)$ *and undefined otherwise.*

Obviously, transformations interact with the validity of graph constraints. Two well-studied notions are constraint-guaranteeing and -preserving transformations [11].

Definition 4 (*c*-guaranteeing and -preserving transformation). *Given a constraint* c*, a transformation* $G \Rightarrow_{r,m} H$ *is* c-guaranteeing *if* $H \models c$*. Such a transformation is* c-preserving *if* $G \models c \Rightarrow H \models c$*. A rule* r *is* c-guaranteeing *(*c-preserving*) if every transformation via* r *is.*

As we will present criteria for consistency sustainment and improvement based on conflicts and dependencies of rules, we recall these notions here as well. Intuitively, a transformation step *causes a conflict* on another one if it hinders this second one. A transformation step is *dependent* on another one if it is first enabled by that.

Definition 5 (Conflict). *Let a pair of transformations* $(t_1, t_2) : (G \Rightarrow_{m_1,r_1} H_1, G \Rightarrow_{m_2,r_2} H_2)$ *applying rules* $r_i = (L_i \hookleftarrow K_i \hookrightarrow R_i, ac_i)$*,* $i = 1, 2$ *be given such that* t_i *yields transformation morphisms* $G \overset{g_i}{\leftarrow} D_i \overset{h_i}{\rightarrow} H_i$*.* Transformation pair (t_1, t_2) *is* conflicting *(or* t_1 causes a conflict on t_2*) if there does not exist a morphism* $x : L_2 \to D_1$ *such that* $g_1 \circ x = m_2$ *and* $h_1 \circ x \models ac$*.* Rule pair (r_1, r_2) *is* conflicting *if there exists a conflicting transformation pair* $(G \Rightarrow_{m_1,r_1} H_1, G \Rightarrow_{m_2,r_2} H_2)$*. If* (r_1, r_2) *and* (r_2, r_1) *are both not conflicting, rule pair* (r_1, r_2) *is called* parallel independent*.*

Definition 6 (Dependency). *Let a sequence* $t_1; t_2 : G \Rightarrow_{m_1,r_1} H_1 \Rightarrow_{m_2,r_2} X$ *of transformations applying rules* $r_i = (L_i \hookleftarrow K_i \hookrightarrow R_i, ac_i)$*,* $i = 1, 2$ *be given such that* t_1 *yields transformation morphisms* $G \overset{g_1}{\leftarrow} D_1 \overset{h_1}{\rightarrow} H_1$*. Transformation* t_2 *is* dependent on t_1 *if there does not exist a morphism* $x : L_2 \to D_1$ *such that* $h_1 \circ x = m_2$ *and* $g_1 \circ x \models ac_2$*. Rule* r_2 *is* dependent on *rule* r_1 *if there exists a transformation sequence* $t_1; t_2 : G \Rightarrow_{m_1,r_1} H_1 \Rightarrow_{m_2,r_2} X$ *such that* t_2 *is dependent on* t_1*. If* r_1 *is not dependent on* r_2 *and* r_2 *is not dependent on* r_1*, rule pair* (r_1, r_2) *is called* sequentially independent*.*

A weak critical sequence *is a sequence* $t_1; t_2 : G \Rightarrow_{m_1,r_1} H_1 \Rightarrow_{m_2,r_2} X$ *of transformations such that* t_2 *depends on* t_1*,* n_1 *and* m_2 *are jointly surjective (where* n_1 *is the co-match of* t_1*), and* m_i *is* not *required to satisfy* ac_i *(*$i = 1, 2$*).*

As rule r_2 in a rule pair (r_1, r_2) will always be plain in this paper, a transformation step can cause a conflict on another one if and only if it deletes an element that the second transformation step matches. Similarly, a transformation step can depend on another one if and only if the first step creates an element that the second matches or deletes an edge that is adjacent to a node the second one deletes.

4 Consistency-sustaining and consistency-improving rules and transformations

In this section, we introduce our key new concepts. We do so in three stages, first introducing foundational definitions for partial consistency, followed by a generic definition of consistency sustainment and improvement. Finally, we give stronger definitions for which we will be able to provide a static analysis in Sect. 5.

4.1 Partial consistency

To support the discussion and analysis of rules and transformations that improve graph consistency, but do not produce a fully consistent graph in one step, we introduce the notion of *partial consistency.* We base this notion on relating the number of *constraint violations* to the total number of *relevant occurrences* of a constraint. For the satisfaction of an existential constraint, a single valid occurrence is enough. In contrast, universal constraints require the satisfaction of some sub-constraint for every occurrence. Hence, the resulting notion is binary in the existential case, but graduated in the universal one.

In the remainder of this paper, a *constraint* is always a linear constraint in ANF having a nesting level ≥ 1.[4] Moreover, all graphs are finite.

Definition 7 (Occurrences and violations). *Let* $c = Q(\emptyset \to C, d)$ *with* $Q \in \{\exists, \forall\}$ *be a constraint. An* occurrence *of* c *in a graph* G *is an injective morphism* $p : C \hookrightarrow G$*, and* $occ(G, c)$ *denotes the* number of such occurrences*.*

If c *is universal, its number of* relevant occurrences *in a graph* G*, denoted as* $ro(G, c)$*, is defined as* $ro(G, c) := occ(G, c)$ *and its* number of constraint violations*, denoted as* $ncv(G, c)$*, is the number of occurrences* p *for which* $p \not\models d$*.*

If c *is existential,* $ro(G, c) := 1$ *and* $ncv(G, c) := 0$ *if there exists an occurrence* $p : C \hookrightarrow G$ *such that* $p \models d$ *but* $ncv(G, c) := 1$ *otherwise.*

[4] Requiring nesting level ≥ 1 is no real restriction as constraints with nesting level 0 are Boolean combinations of `true` which means they are equivalent to `true` or `false`, anyhow. In contrast, restricting to linear constraints actually excludes some interesting cases. We believe that the extension of our definitions and results to also include the non-linear case will be doable. Restricting to the linear case first, however, makes the statements much more accessible and succinct.

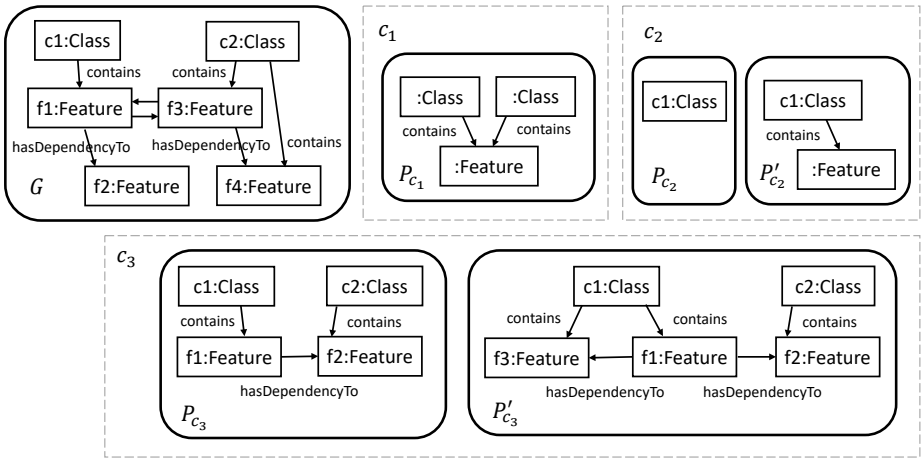


Fig. 3. Example constraints and graph.

Definition 8 (Partial consistency). *Given a graph G and a constraint c, G is* consistent *w.r.t. c if $G \models c$. The* consistency index of G w.r.t. c *is defined as*

$$ci(G,c) \coloneqq 1 - \frac{ncv(G,c)}{ro(G,c)}$$

where we set $\frac{0}{0} \coloneqq 0$. We say that G is partially consistent *w.r.t. c if $ci(G,c) > 0$.*

The next proposition makes precise that the consistency index runs between 0 and 1 and indicates the degree of consistency a graph G has w.r.t. a constraint c.

Fact 1 (Consistency index). *Given a graph G and a constraint c, then $0 \leq ci(G,c) \leq 1$ and $G \models c$ if and only if $ci(G,c) = 1$. Consistency implies partial consistency. Moreover, $ci(G,c) \in \{0,1\}$ for an existential constraint.*

Example 1. Based on Fig. 3, we can express the three informal constraints from Section 2 as nested graph constraints. Constraint c_1 can be expressed as $\neg\exists P_{c_1}$, constraint c_2 becomes $\forall(P_{c_2}, \exists P'_{c_2})$, and constraint c_3 becomes $\forall(P_{c_3}, \exists P'_{c_3})$. Graph G (in the left top corner of Fig. 3) satisfies c_1 and c_2. It does not satisfy c_3, since we cannot find an occurrence of P'_{c_3} for the occurrence of P_{c_3} in G where *f1* and *f2* are mapped to *f1* and *f3*, respectively. Graph G in Fig. 3 has the consistency index 0.5 with regard to c_3, since one violation exists, and two non-violating occurrences are required.

4.2 Consistency sustainment and improvement

In the remainder of this section, our goal is to introduce the notions of *consistency-sustaining* and *consistency-improving rule applications* which refine the established notions of preserving and guaranteeing applications [11].

Definition 9 (Consistency sustainment and improvement). *Given a graph constraint c and a rule r, a transformation $t : G \Rightarrow_{r,m} H$ is* consistency sustaining *w.r.t. c if $ci(G, c) \leq ci(H, c)$. It is* consistency improving *if it is consistency sustaining, $ncv(G, c) > 0$, and $ncv(G, c) > ncv(H, c)$.*

The rule r is consistency sustaining *if all of its applications are. It is* consistency improving *if all of its applications are consistency sustaining and there exists a graph $G \in Graph_{TG}$ with $ncv(G, c) > 0$ and a consistency-improving transformation $G \Rightarrow_{r,m} H$. A consistency improving rule is* strongly consistency improving *if all of its applications to graphs G with $ncv(G, c) > 0$ are consistency-improving transformations.*

In the above definition, we use the number of constraint violations (and not the consistency index) to define improvement to avoid an undesirable side-effect: Defining improvement via a growing consistency index would lead to consistency-improving transformations (w.r.t. a universal constraint) which do not repair existing violations but only create new valid occurrences of the constraint. Hence, there would exist infinitely long transformation sequences where every step increases the consistency index but validity is never restored. Consistency-improving transformations, and therefore *strongly* consistency improving rules, require that the number of constraint violations strictly decreases in each step. Therefore, using only such transformations and rules, we cannot construct infinite transformation sequences.

Any consistency-improving rule can be turned into a strongly consistency-improving rule if suitable pre-conditions can be added that restrict the applicability of the rule only to those cases where it can actually improve a constraint violation. This links the two forms of consistency-improving rules to their practical applications: in model repair [21,25] we want to use rules that will only make a change to a graph when there is a violation to be repaired—strongly consistency-improving rules. However, in evolutionary search [5], we want to allow rules to be able to make changes even when there is no need for repair, but to fix violations when they occur; consistency-improving rules are well-suited here as they can be applied even when no constraint violations need fixing.

4.3 Direct consistency sustainment and improvement

While the above definitions are easy to state and understand, it turns out that they are inherently difficult to investigate. Comparing numbers of (relevant) occurrences and violations allows for very disparate behavior of consistency-sustaining (-improving) transformations: For example, a transformation is allowed to destroy as many valid occurrences as it repairs violations and is still considered to be consistency sustaining w.r.t. a universal constraint.

Next, we introduce further qualified notions of consistency sustainment and improvement. The idea behind this refinement is to *retain* the validity of occurrences of a universal constraint: valid occurrences that are preserved by a transformation are to remain valid. In this way, sustainment and improvement become more *direct* as it is no longer possible to compensate for introduced

violations by introducing additional valid occurrences. The notions of (direct) sustainment and improvement are related to one another and also to the already known ones that preserve and guarantee constraints. In Sect. 5 we will show how these stricter definitions allow for static analysis techniques to identify consistency-sustaining and -improving rules.

The following definitions assume a transformation step to be given and relate occurrences of constraints in its start and result graph as depicted in Fig. 4. The existence of a morphism p_D such that the left triangle commutes (and p' might be defined as $h \circ p_D$) is equivalent to the tracking morphism $tr : G \dashrightarrow H$ being a total morphism when restricted to $p(C)$ which is equivalent to the transformation not destroying the occurrence p.

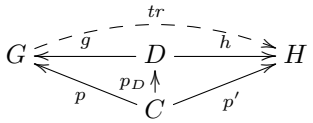


Fig. 4. Rule application with morphisms from a graph C, occurring in some constraint

Definition 10 (Direct consistency sustainment). *Given a graph constraint c, a transformation $t : G \Rightarrow_{m,r} H$ via rule r at match m with trace tr (Fig. 4) is* directly consistency sustaining *w.r.t. c if either c is existential and the transformation is c-preserving or $c = \forall(C, d)$ is universal and*

$$\forall p : C \hookrightarrow G\big((p \models d \wedge tr \circ p \text{ is total}) \Rightarrow tr \circ p \models d\big) \wedge$$
$$\forall p' : C \hookrightarrow H\big(\neg\exists p : C \hookrightarrow G\,(p' = tr \circ p) \Rightarrow p' \models d\big) \ .$$

A rule *r is* directly consistency sustaining *w.r.t. c if all its applications are.*

The first requirement in the definition checks that constraints that were already valid in G are still valid in H, unless their occurrence has been removed; that is, the transformation must not make existing valid occurrences invalid. Note, however, that we do not require that the constraint be satisfied by the same extension, just that there is still a way to satisfy the constraint at that occurrence. The second requirement in the definition checks that every "new" occurrence of the constraint in H satisfies the constraint; that is, the transformation must not introduce fresh violations.

Table 1. Properties of example rules.

	Consistency sustaining			**Consistency improving**		
Rule	c_1	c_2	c_3	c_1	c_2	c_3
assignFeature	+	+	-	-	+	-
createClass	+	+	-	-	-	-
moveFeature	(+)	-	-	-	-	-
deleteEmptyClass	+	+	+	-	+*	-

Legend: + denotes *directly*, (+) denotes *non-directly*, * denotes *strongly*

The following theorem relates the new notions of (direct) consistency sustainment to preservation and guarantee of constraints.

Theorem 2 (Sustainment relations). *Given a graph constraint c, every c-guaranteeing transformation is directly consistency-sustaining, every directly consistency-sustaining transformation is consistency sustaining, and every consistency-sustaining transformation is c-preserving. The analogous implications hold on the rule level:*

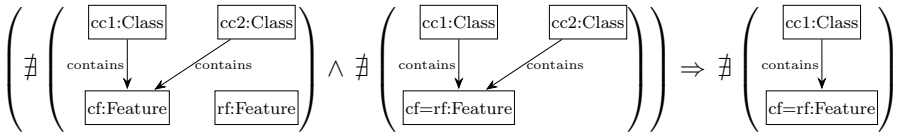
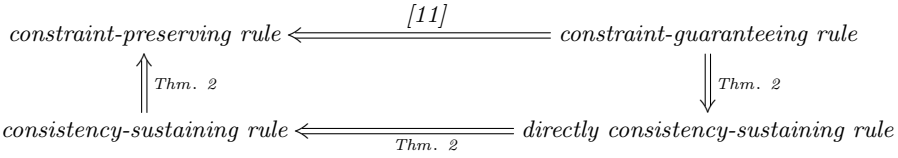


Fig. 5. Generated preserving application condition for *createClass* w.r.t. constraint c_1. The feature named `rf` is the one from the LHS of *createClass*.



The following example illustrates these notions and shows that sustainment is different from constraint guaranteeing or preserving.

Example 2. Table 1 denotes for each rule from the running example if it is consistency sustaining w.r.t. each constraint. Rule *createClass* is directly consistency sustaining w.r.t. c_1 (no double assignments) and c_2 (no empty classes), since it cannot assign an already assigned feature or remove existing assignments. However, it is not consistency guaranteeing, since it cannot remove any violation either. Rule *moveFeature* is consistency sustaining w.r.t c_1, but not directly so, since it can introduce new violations, but only while at the same time removing another violation, leading to a neutral outcome. Starting with the plain version of rule *createClass* and computing a preserving application condition for constraint c_1 according to the construction provided by Habel and Pennemann [11] results in the application condition depicted in Fig. 5. By construction, equipping the plain version of *createClass* with that application condition results in a consistency-preserving rule. However, whenever applied to an invalid graph, the antecedent of this application condition evaluates to `false` and, hence, the whole application condition to `true`. In particular, the rule with this application condition might introduce further violations of c_1 and is, thus, not sustaining.

Similarly, the *direct* notion of consistency improvement preserves the validity of already valid occurrences in the case of universal constraints and degenerates to the known concept of constraint-guarantee in the existential case.

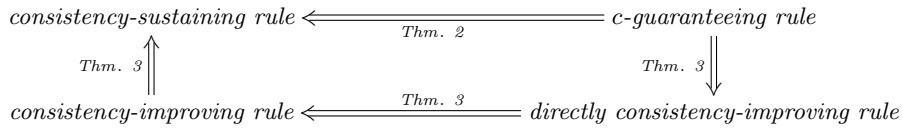
Definition 11 (Direct consistency improvement). *Given a graph constraint c, a transformation $t : G \Rightarrow_{m,r} H$ via rule r at match $m : L \hookrightarrow G$ with trace tr (Fig. 4) is* directly consistency improving *w.r.t. c if $G \not\models c$, the transformation is directly consistency sustaining, and either c is existential and the transformation is c-guaranteeing or $c = \forall(C, d)$ is universal and*

$$\exists p : C \hookrightarrow G\big(p \not\models d \wedge p' := tr \circ p \text{ is total} \wedge p' \models d\big) \vee$$
$$\exists p : C \hookrightarrow G\big(p \not\models d \wedge p' := tr \circ p \text{ is not total}\big)$$

We lift the notion of directly consistency-improving transformations to the level of rules in the same way as in Def. 9. This leads to directly consistency-improving rules and a strong form of directly consistency-improving rules.

(Direct) consistency improvement is related to, but different from constraint guarantee and consistency sustainment as made explicit in the next theorem.

Theorem 3 (Improvement relations). *Given a graph constraint c, every directly consistency-improving transformation is a consistency-improving transformation and every consistency-improving transformation is consistency sustaining. Moreover, every c-guaranteeing transformation starting from a graph G that is inconsistent w.r.t. c is a directly consistency-improving transformation. The analogous implications hold on the rule level, provided that there exists a match for the respective rule r in a graph G with $G \not\models c$:*



Example 3. Table 1 denotes for each rule of the running example if it is consistency improving w.r.t. each constraint. For example, the rule *deleteEmptyClass* is directly strongly consistency improving but not guaranteeing w.r.t. c_2 (no empty classes), since it always removes a violation (empty class), but generally not all violations in one step. Rule *assignFeature* is consistency improving w.r.t. c_2, but not directly so, since it can turn empty classes into non-empty ones, but does not do so in every possible application. Rule *createClass* is consistency sustaining but not improving w.r.t. c_2, as it cannot reduce the number of empty classes.

5 Static Analysis for Direct Consistency Sustainment and Improvement

In this section, we consider specific kinds of constraints and present a static analysis technique for direct consistency sustainment and improvement. We present criteria for rules to be *directly* consistency sustaining or *directly* consistency improving w.r.t. these kinds of constraint. The restriction to specific kinds of constraint greatly simplifies the presentation; at the end of the section we hint at how our results may generalize to arbitrary universal constraints.

The general idea behind our static analysis technique is to check for validity of a constraint by applying a trivial (non-modifying) rule that just checks for the existence of a graph occurring in the constraint. This allows us to present our analysis technique in the language of *conflicts and dependencies* which has been developed to characterise the possible interactions between rule applications [24,8]. As a bonus, since the efficient detection of such conflicts and dependencies has been the focus of recent theoretical and practical research [17,18], we obtain tool support for an automated analysis based on Henshin.

In the remainder of this paper, we assume the following setting: Let $r = (L \hookleftarrow K \hookrightarrow R, ac)$ be a rule, c a graph constraint of the form $\neg\exists C = \forall(\emptyset \hookrightarrow C, \texttt{false})$ and d a graph constraint of the form $\forall(C, \exists C') = \forall(\emptyset \hookrightarrow C, \exists a : C \hookrightarrow C')$. Given a graph G, there is the rule $check_G \coloneqq G \xleftarrow{id_G} G \xhookrightarrow{id_G} G$ given.

For the statement of the following results, note that sequential independence of the (non-modifying) rule $check_C$ from r means that r cannot create a new match for C. Similarly, parallel independence of $check_{C'}$ from r means that r cannot destroy a match for C'. We first state criteria for direct consistency sustainment: If a rule cannot create a new occurrence of C, it is directly consistency sustaining w.r.t. a constraint of the form $\neg\exists C$. If, in addition, it cannot delete an occurrence of C', it is directly consistency sustaining w.r.t. a constraint of the form $\forall(C, \exists C')$.

Theorem 4 (Criteria for direct consistency sustainment). *Rule r is directly consistency sustaining w.r.t. constraint c if and only if $check_C$ is sequentially independent from r. If, in addition, $check_{C'}$ is parallel independent from r, then r is directly consistency sustaining w.r.t. constraint d.*

The above criterion is sufficient but not necessary for constraints of the form $\forall(C, \exists C')$. For example, it does not take into account the possibility of r creating a new valid occurrence of C. The next proposition strengthens the above theorem by partially remedying this.

Proposition 1. *If $check_{C'}$ is parallel independent from r and for every weak critical sequence $G \Rightarrow_{r,m} H \Rightarrow_{check_C, p''} H$ it holds that there is an injective morphism $q'' : C' \hookrightarrow H$ with $q'' \circ a = p''$, i.e., $p'' \models \exists C'$, then r is directly consistency sustaining w.r.t. constraint d.*

For consistency improvement we state criteria on rules as well: If a rule is directly consistency improving w.r.t. a constraint of the form $\forall(C, \exists C')$, it is either (1) able to destroy an occurrence of C (deleting a part of it) or (2) to bring about an occurrence of C' (creating a part of it). In case (2), we can even be more precise: The newly created elements do not stem from C but from the part of C' without C; this is what the formula in the next theorem expresses. For constraints of the form $\neg\exists C$, condition (1) is the only one that holds.

Theorem 5 (Criteria for direct consistency improvement). *If rule r is directly consistency sustaining w.r.t. constraint c, then it is directly consistency improving w.r.t. c if and only if r causes a conflict for $check_C$. If r is directly consistency improving w.r.t. constraint d, then r causes a conflict for $check_C$ or $check_{C'}$ is sequentially dependent on r in such a way that*

$$n(R \setminus K) \cap p'(C') \subseteq p'(C' \setminus a(C))$$

where, in this dependency, n is the co-match of the first transformation applying r and p' is the match for $check_{C'}$.

Table 2. Generalisation of the criteria from Theorems 4 and 5 to universal constraints up to nesting level 2. Here, ck_C is short for $check_C$, $r_1 <_D r_2$ denotes dependency of r_2 on r_1, $r_1 <_C r_2$ denotes r_2 causing a conflict for r_1, and crossed out versions denote the respective absence.

type of constr.	**crit. for direct consist. sust.**	**crit. for direct consist. impr.**
$\forall(C, \texttt{false}) \equiv \neg\exists C$	$ck_C \not<_D r$	$ck_C <_C r$
$\forall(C_1, \exists C_2)$	$ck_{C_1} \not<_D r \wedge ck_{C_2} \not<_C r$	$ck_{C_1} <_C r \vee ck_{C_2} <_D r$
$\forall(C_1, \exists(C_2, \neg\exists C_3))$	$ck_{C_1} \not<_D r \wedge ck_{C_2} \not<_C r \wedge ck_{C_3} \not<_D r$	$ck_{C_1} <_C r \vee ck_{C_2} <_D r \vee ck_{C_3} <_C r$

Table 3. Applying the criteria from Tbl. 2 to the example; ck_C is short for $check_C$.

	Consis. sust. (suff. cr.)					**Consis. impr. (necc. cr.)**				
	seq. indep.			par. indep.		par. dep.			seq. dep.	
Rule	$ck_{P_{c_1}}$	$ck_{P_{c_2}}$	$ck_{P_{c_3}}$	$ck_{P'_{c_2}}$	$ck_{P'_{c_3}}$	$ck_{P_{c_1}}$	$ck_{P_{c_2}}$	$ck_{P_{c_3}}$	$ck_{P'_{c_2}}$	$ck_{P'_{c_3}}$
assignFeature	-	+	-	+	+	-	-	-	+	+
createClass	-	-	-	+	+	-	-	-	+	+
moveFeature	-	+	-	-	-	+	-	+	+	+
deleteEmptyClass	+	+	+	+	+	-	+	-	-	-

The above criterion is not sufficient in case of constraint d. The existing conflicts or dependencies do not ensure that actually an *invalid* occurrence of C can be deleted or a new occurrence of C' can be created in such a way that an invalid occurrence of C is "repaired".

Looking closer to the criteria stated above, we can find some recurring patterns. Table 2 lists the kinds of universal constraints up to nesting level 2 and the corresponding criteria. While we have shown the criteria in the first two rows in Theorems 4 and 5, we conjecture the criteria in the last row of Table 2. To prove generalized theorems for nesting levels ≥ 2, however, is up to future work.

Example 4. We can use the criteria in Table 2 to semi-automatically reason about consistency sustainment and improvement in our example. To this end, we first apply automated conflict and dependency analysis (CDA, [18]) to the relevant pairs of mutation and check rules. Using the detected conflicts and dependencies, we infer parallel and sequential (in)dependence per definition, as shown in Table 3. For example, since no dependencies between *assignFeature* and $check_{P_{c_1}}$ exist, we conclude that these rules are sequentially independent.

Consistency sustainment: Based on Table 3, we find that the sufficient criterion formulated in Theorem 4 is adequate to show direct consistency sustainment in four out of seven positive cases as per Table 1: rule *assignFeature* with constraint c_3 and rule *deleteEmptyClass* with constraints c_1, c_2 and c_3. Moreover, the stronger criterion in Proposition 1 allows to recognize the case of *createClass* with c_2. Discerning the remaining two positive cases (*assignFeature* with c_1; *createClass* with c_1) from the five negative ones requires further inspection.

Consistency improvement: Based on Table 3, our necessary criterion allows to detect the two positive cases in Table 1: rules *deleteEmptyClass* and *assign-*

Feature with constraint c_2. The former is due to parallel dependence, the latter due to sequential dependence (where inspection of the CDA results reveals a critical sequence with a suitable co-match). The criterion is also fulfilled in six negative cases: *assignFeature* with c_3, *createClass* with c_2 and c_3, and *moveFeature* with c_1, c_2 and c_3. Four negative cases are correctly ruled out by the criterion.

6 Related Work

In this paper, we introduce a graduated version of a specific logic on graphs, namely of nested graph constraints. Moreover, we focus on the interaction of this graduation with graph transformations. Therefore, we leave a comparison with fuzzy or multi-valued logics (on graphs) to future work. Instead, we focus on works that also investigate the interaction between the validity of nested graph constraints and the application of transformation rules.

Given a graph transformation (sequence) $G \Rightarrow H$, the validity of graph H can be established with basically three strategies: (1) graph G is already valid and this validity is preserved, (2) graph G is not valid and there is a c-guaranteeing rule applied, and (3) graph G is made valid by a graph transformation (sequence) step-by-step.

Strategies (1) and (2) are supported by the incorporation of constraints in application conditions of rules as presented in [11] for nested graph constraints in general and implemented in Henshin [19]. As the applicability of rules enhanced in that way can be severely restricted, improved constructions have been considered of specific forms of constraints. For constraints of the form $\forall(C, \exists C')$, for example, a suitable rule scheme is constructed in [15]. In [2] refactoring rules are checked for the preservation of constraints of nesting level ≤ 2. In [19], two of the present authors also suggested certain simplifications of application conditions; the resulting ones are still constraint-preserving. In [20], we even showed that they result in the logically weakest application condition that is still directly consistency sustaining. However, the result is only shown for negative constraints of nesting level one. A very similar construction of negative application conditions from such negative constraints has very recently been suggested in [3].

Strategy (3) is followed in most of the rule-based graph repair or model repair approaches. In [22], the violation of mainly multiplicity constraints is considered. In [12], Habel and Sandmann derive graph programs from graph constraints of nesting level ≤ 2. In [25], they extend their results to constraints in ANF which end with $\exists C$ or constraints of one of the forms $\exists(C, \neg\exists C')$ or $\neg\exists C$. They also investigate whether a given set of rules allows to repair such a given constraint. In [7] Dyck and Giese present an approach to automatically check whether a transformation sequence yields a graph that is valid w.r.t. specific constraints of nesting level ≤ 2.

Up to now, result graphs of transformations have been considered either valid or invalid w.r.t. to a graph constraint; intermediate consistency grades have not been made explicit. Thereby, c-preserving and c-guaranteeing transfor-

mations [11] focus on the full validity of the result graphs. Our newly developed notions of consistency-sustainment and improvement are located properly in between existing kinds of transformations (as proven in Theorems 2 and 3). These new forms of transformations make the gradual improvements in consistency explicit. While a detailed and systematic investigation (applying the static methods developed in this paper) is future work, a first check of the kinds of rules generated and used in [14] (model editing), [22] (model repair), and [5] (search-based model engineering) reveals that—in each case—at least some of them are indeed (directly) consistency-sustaining. We are therefore confident that the current paper formalizes properties of rules that are practically relevant in diverse application contexts. Work on partial graphs as in, e.g. [26], investigates the validity of constraints in families of graphs which is not our focus here and therefore, not further considered.

Stevens in [27] discusses similar challenges in the specific context of bidirectional transformations. Here, consistency is a property of a pair of models (loosely, graphs) rather than between a graph and constraint. In this sense, it may be argued that our formalisation generalises that of [27]. Several concepts are introduced that initially seem to make sense only in the specific context of bidirectional transformations (*e.g.,* the idea of $\vec{R}$ candidates), but may provide inspiration for a further extension of our framework with corresponding concepts.

7 Conclusions

In this paper, we have introduced a definition of graph consistency as a graduated property, which allows for graphs to be partially consistent w.r.t. a nested graph constraint, inducing a partial ordering between graphs based on the number of constraint violations they contain. Two new forms of transformation can be identified as consistency sustaining and consistency improving, respectively. They are properly located in between the existing notions of constraint-preserving and constraint-guaranteeing transformations. Lifting them to rules, we have presented criteria for determining whether a rule is consistency sustaining or improving w.r.t. a graph constraint. We have demonstrated how these criteria can be applied in the context of a case study from search-based model engineering.

While the propositions we present allow us to check a given rule against a graph constraint, their lifting to a set of constraints is the next step to go. Furthermore, algorithms for constructing consistency-sustaining or -improving rules from a set of constraints are left for future work.

Acknowledgements. We thank the ICGT reviewers for their insightful and helpful comments. This work has been partially supported by DFG grants TA 294/17-1 and 413074939.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Proc. MODELS. pp. 121–135. Springer (2010)
2. Becker, B., Lambers, L., Dyck, J., Birth, S., Giese, H.: Iterative Development of Consistency-Preserving Rule-Based Refactorings. In: ICMT. pp. 123–137. Springer, Berlin (2011)
3. Behr, N., Saadat, M.G., Heckel, R.: Commutators for Stochastic Rewriting Systems: Theory and Implementation in Z3 (2020), `https://arxiv.org/abs/2003.11010`
4. Bowman, M., Briand, L.C., Labiche, Y.: Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE Transactions on Software Engineering **36**(6), 817–837 (2010)
5. Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic consistency preserving search operators for search-based model engineering. In: MODELS. pp. 106–116. IEEE (2019)
6. Burdusel, A., Zschaler, S., Strüber, D.: MDEOptimiser: A search based model engineering tool. In: MODELS. pp. 12–16 (2018)
7. Dyck, J., Giese, H.: k-inductive invariant checking for graph transformation systems. In: Graph Transformation - 10th International Conference, ICGT 2017. LNCS, vol. 10373, pp. 142–158. Springer (2017)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
9. Fleck, M., Troya, J., Wimmer, M.: Marrying search-based optimization and model transformation technology. In: NasBASE (2015)
10. Fleck, M., Troya Castilla, J., Wimmer, M.: The class responsibility assignment case. TTC (2016)
11. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. in Comp. Science **19**, 245–296 (2009)
12. Habel, A., Sandmann, C.: Graph Repair by Graph Programs. In: STAF. pp. 431–446. Springer, Cham (2018)
13. Jensen, M.T.: Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation. Journal of Mathematical Modelling and Algorithms **3**(4), 323–347 (2004)
14. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: ICMT. pp. 173–188. Springer, Cham (2016)
15. Kosiol, J., Fritsche, L., Nassar, N., Schürr, A., Taentzer, G.: Constructing Constraint-Preserving Interaction Schemes in Adhesive Categories. In: WADT. pp. 139–153. Springer (2019)
16. Kosiol, J., Strüber, D., Taentzer, G., Zschaler, S.: Graph Consistency as a Graduated Property: Consistency-Sustaining and -Improving Graph Transformations – Extended Version (2020), `https://arxiv.org/abs/2005.04162`
17. Lambers, L., Born, K., Kosiol, J., Strüber, D., Taentzer, G.: Granularity of conflicts and dependencies in graph transformation systems: A two-dimensional approach. J. Log. Algebr. Meth. Program. **103**, 105–129 (2019)
18. Lambers, L., Strüber, D., Taentzer, G., Born, K., Huebert, J.: Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In: ICSE. pp. 716–727. ACM (2018)

19. Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: Constructing optimized validity-preserving application conditions for graph transformation rules. In: ICGT. pp. 177–194. Springer (2019)
20. Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: Constructing optimized validity-preserving application conditions for graph transformation rules. Journal of Logical and Algebraic Methods in Programming (2020), (to appear)
21. Nassar, N., Kosiol, J., Radke, H.: Rule-based Repair of EMF Models: Formalization and Correctness Proof. In: GCM (2017)
22. Nassar, N., Radke, H., Arendt, T.: Rule-based repair of EMF models: An automated interactive approach. In: ICMT. pp. 171–181. Springer, Cham (2017)
23. Nuseibeh, B., Easterbrook, S., Russo, A.: Making inconsistency respectable in software development. Journal of Systems and Software **58**(2), 171–180 (2001)
24. Plump, D.: Confluence of graph transformation revisited. In: Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday. pp. 280–308. Springer (2005)
25. Sandmann, C., Habel, A.: Rule-based graph repair. CoRR **abs/1912.09610** (2019), `http://arxiv.org/abs/1912.09610`
26. Semeráth, O., Varró, D.: Graph constraint evaluation over partial models by constraint rewriting. In: ICMT. pp. 138–154. Springer (2017)
27. Stevens, P.: Bidirectionally tolerating inconsistency: Partial transformations. In: Gnesi, S., Rensink, A. (eds.) Int'l Conf. Fundamental Approaches to Software Engineering (FASE'14). pp. 32–46. Springer Berlin Heidelberg (2014)
28. Strüber, D.: Generating efficient mutation operators for search-based model-driven engineering. In: ICMT. pp. 121–137. Springer (2017)
29. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for EMF model transformation development. In: ICGT. pp. 196–208. Springer (2017)