Journal of Logical and Algebraic Methods in Programming 114 (2020) 100564

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Constructing optimized constraint-preserving application conditions for model transformation rules ☆

Nebras Nassar *, Jens Kosiol, Thorsten Arendt, Gabriele Taentzer

Philipps-Universität Marburg, Marburg, Germany

ARTICLE INFO

Article history:
Received 14 December 2019
Received in revised form 18 May 2020
Accepted 21 May 2020
Available online 27 May 2020

Keywords:
Graph transformation
Model transformation
Constraints
Correctness
Validity preservation

ABSTRACT

There is an increasing need for model transformations ensuring valid result models w.r.t. a given constraint. In model refactoring, for example, each performed refactoring should yield a valid model again. Given a constraint, if a model transformation rule always produces valid output, it is called *constraint-guaranteeing*; if only when applied to an already valid model, it is called *constraint-preserving*. In the literature, there is a formal construction for model transformation systems making them constraint-guaranteeing. This is ensured by adding application conditions to their transformation rules. These conditions can become quite large, though. As there are interesting application cases where transformations just need to be constraint-preserving (such as model refactoring), the construction of application conditions was also adapted to this case. Although logically weaker, the straightforward construction can lead to even larger application conditions. In this work, we develop simplifications of constraint-guaranteeing conditions by omitting certain parts of these conditions, namely of parts that check for antecedent validity. We prove that the resulting application conditions are constraint-preserving and characterize their logical strength. Our theory is developed for $\mathcal{M}$-adhesive categories which encompass various graph-like model structures. In addition, the computation of constraint-guaranteeing application conditions and their simplifications was implemented in the Eclipse plug-in OCL2AC. Evaluations show that the complexity of the constructed simplified conditions is reduced by factor 7 on average. Moreover, this optimization yields a speedup of rule application by approximately 2.5 times.

1. Introduction

Model transformations are the heart and soul of Model-Driven Engineering (MDE). They are used for various MDE-activities including translation, optimization, and synchronization of models [1]. Usually, a transformation (that may consist of several transformation steps) should yield a valid result model, especially if it has been applied to an already valid model. Intermediate models may not be required to be valid; this has been argued generally for software systems [2] and has been approved for modeling in, e.g., [3]. But there are scenarios where even intermediate models have to show validity,

☆ This work was partially funded by the German Research Foundation (DFG), projects "Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages" (grant no. TA 294/13-2) and "Triple Graph Grammars (TGG) 2.0" (grant no. TA 294/17-1).

* Corresponding author.
E-mail addresses: nassarn@informatik.uni-marburg.de (N. Nassar), kosiolje@mathematik.uni-marburg.de (J. Kosiol), thorsten.arendt@uni-marburg.de (T. Arendt), taentzer@mathematik.uni-marburg.de (G. Taentzer).

https://doi.org/10.1016/j.jlamp.2020.100564
2352-2208/

at least a basic one, as the following example applications show: (1) Throughout a larger refactoring process, each performed refactoring should preserve the model's validity [4]. (2) More generally, any in-place model change should preserve a basic validity, enough to view an edited model in its domain-specific model editor [5]. Model editors typically ensure the creation of models with basic validity right from the beginning. This is the application scenario we will use as running example and for our evaluation. A similar scenario is considered in projectional editing for textual editors [6]. (3) Modeling the behavior of concurrent and distributed systems with model transformations, each model represents a system state that should fulfill system invariants such as safety properties [7]. (4) When generating code from abstractly specified model transformations, the transformations should be constraint-preserving, especially for safety-critical systems [8].

State of the art. From the formal point of view, the theory of algebraic graph transformation constitutes a suitable framework to reason about model transformations [9,10], in particular about rule-based transformation of EMF models [11]. Constraints are typically expressed as (nested) graph constraints [12,13], into which a large and relevant part of OCL [14] can be translated [15]. Graph constraints can be integrated as application conditions into graph transformation rules as shown in [13]: Given a rule and a constraint, there are two variants of integration, namely computing a *constraint-preserving* or a *constraint-guaranteeing* rule. Both computations do not alter the actions of the rule but equip it with application conditions. Graph validity is *preserved* if, applying an equipped rule to a valid graph, the resulting graph is valid as well. Graph validity is *guaranteed* if, applying an equipped rule to any graph, the resulting graph is valid. Besides, very recently the notion of *(direct) sustainment* of consistency has been introduced. It formalizes the idea that the application of a rule does not increase the amount of "invalidity" of a graph with respect to a given constraint. Sustainment is shown to imply preservation of the constraint in question [16].

Whereas no construction for sustaining rules has been presented, the known ones for preservation and guarantee have certain complexity issues. Theoretically, the number of graphs in a constraint-guaranteeing application condition can grow exponentially [17]. As there are interesting application cases where transformations just need to be constraint-preserving (as pointed out above), it is worthwhile to investigate this case further. Although logically weaker, following the construction in the literature, constraint-preserving application conditions may contain even more elements than corresponding constraint-guaranteeing ones. This is due to the approach taken: The premise that the model was already valid before rule application is added to the computed constraint-guaranteeing application condition. The resulting condition can be inherently difficult to simplify because of the used material implication operator.

Contributions. In this paper, we develop optimizing-by-construction techniques in the general context of $\mathcal{M}$-adhesive categories [18,10] to construct application conditions that preserve validity and present tool support to compute constraint-guaranteeing application conditions as well as the optimized constraint-preserving ones. We formally show the correctness of our approach, discuss the architecture of our tool, and empirically show that the resulting application conditions are considerably less complex which also leads to a faster application of the equipped rules.

In more detail: We take a constraint and a rule as starting point and construct an application condition that preserves validity. This construction is based on the construction of the constraint-guaranteeing application condition but simplifies it by omitting parts that check for antecedent validity, while keeping parts that prevent the introduction of violations. A first class of such possible simplifications is introduced in the general framework of so-called $\mathcal{M}$-adhesive categories, which encompass (typed attributed) graphs in particular. We show that the resulting application condition is not only preserving but a *weakest constraint-preserving application condition* in some and a *weakest consistency-sustaining* one in other cases (Theorem 1). In both cases, this means that they cannot be further simplified from the logical point of view without losing their distinguishing property. We formally compare this result to the construction of constraint-preserving application conditions by Habel and Pennemann [13]. It shows that, in the case of weakest constraint-preserving application conditions, the results of their construction are semantically equivalent to ours; however, theirs are generally far more complex with regard to the condition structure. A second class of simplifications is specific for EMF models and proven to not alter the semantics of the simplified application condition when evaluated on EMF model graphs (Theorem 2); in particular, the properties of being constraint-guaranteeing or -preserving or directly consistency-sustaining are not altered by those simplifications (Corollary 3). We will argue how some of these simplifications omit *global* checks that have to traverse the whole model while keeping *local* ones, i.e., checks being performed in the context of a rule match.

Practically, we have implemented the computation of constraint-guaranteeing application conditions in the tool OCL2AC. It is able to automatically translate OCL constraints into graph constraints and to integrate these as application conditions into transformation rules specified in Henshin [19]. This is the first ready-to-use tool implementing this technique and addressing EMF, i.e., a commonly accepted framework in MDE. On top of the computation of constraint-guaranteeing application conditions, OCL2AC implements the proposed simplifications which result in weakest constraint-preserving or weakest directly consistency-sustaining application conditions. The simplifications take place *during the construction of the constraint-guaranteeing application condition*. It turns out that the computation of the simplified application conditions is faster compared to the computation of the whole constraint-guaranteeing application condition. Comparing the structures of constraint-guaranteeing and -preserving application conditions with that of our simplified ones, empirical results show a considerable loss of structural complexity.

We provide an application case which shows that constraint-preserving transformations are useful in practice. In domain-specific model editing (presented as scenario (2) above), every state of the editing process has to ensure a basic model validity. Each editing step in such a process is specified as rule-based transformation step. The example comprises the

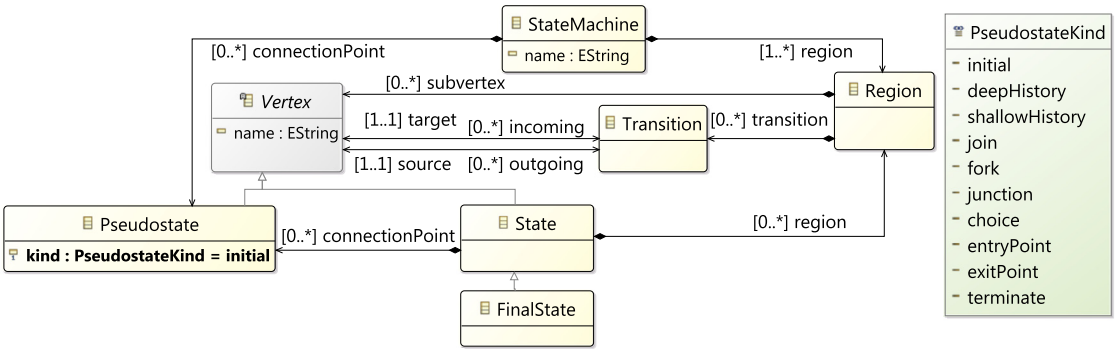


Fig. 1. A simple Statecharts meta-model.

MagicDraw Statechart meta-model with 11 OCL constraints and 84 editing rules. The optimizations do not only reduce the size of computed application conditions considerably but also improve the performance when applying the resulting rules.

In addition, we conducted several evaluations that test the overall approach. We compared the run times of validity checking after a transformation using existing OCL validators (*a posteriori approach*) with running a constraint-preserving transformation (being enriched with application conditions) with and without optimization (*a priori approach*). Results show that both approaches are fast in practice. To the best of our knowledge, this is the first time that the implemented approach (of guaranteeing or preserving validity by the use of application conditions) is empirically investigated.

This paper is an extended version of [20] and in particular of [21]. Beyond these papers, we present the following new contributions:

1. We lift our formal results from graph transformation to $\mathcal{M}$-adhesive categories and present all proofs. This means that our results also cover attributed graphs and further structures.
2. We considerably improve Theorem 1 by not only showing preservation of a constraint but also that the resulting application condition is indeed a weakest constraint-preserving or weakest directly consistency-sustaining application condition.
3. We formally compare to the work of Habel and Pennemann [13].
4. We present more details of OCL2AC by giving an overview of its implementation. The integration of our optimizations is presented in particular.
5. We discuss related work more comprehensively.

Structure. The paper starts with presenting a running example in Sect. 2 and recalling the formal background in Sect. 3. Section 4 presents the theory of all the simplifications of application conditions and compares the results with existing notions. Section 5 presents the tool support. Section 6 contains several evaluations that test our approach and tooling. The longer proofs and two technical lemmas are outsourced to Appendix A.

2. Running example

In this section, we illustrate the effect of our optimizations on application conditions computed by OCL2AC. A simple Statecharts language serves as an example. Its meta-model is displayed in Fig. 1. A StateMachine contains at least one Region and Pseudostates as connection points if they are of kind entryPoint or exitPoint. A Region contains Transitions and Vertices. Vertex is an abstract class with concrete subclasses State and Pseudostate. A State may contain Regions and Pseudostates to support the specification of state hierarchies. FinalState inherits from State. Transitions connect Vertices.

The UML definition specifies several constraints on statechart models. For example, each Transition is required to be contained in a Region (TransitionInRegion) and a FinalState is forbidden to contain a Region (no_Region). Figs. 2 and 3 show

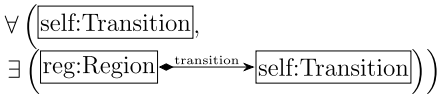


Fig. 2. Graph constraint for TransitionInRegion.

Fig. 3. Graph constraint for no_region.

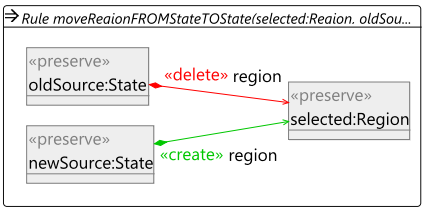


Fig. 4. Transformation rule in Henshin.

these constraints as graph constraints, respectively. The semantics of the first is explained as follows: A model satisfies this constraint if for every Transition (the universally quantified pattern on the first line) there exists a Region containing this Transition via an edge of type transition (the existentially quantified pattern on the second line); the name self identifies the Transitions of the two patterns to be the same. The second constraint just forbids a certain pattern to exist in a model, namely, a FinalState that contains a Region. In the UML, however, these constraints are specified in OCL [14]; the OCL constraint for no_region, for example, is specified as

```
context FinalState invariant no_region: self.region->isEmpty()
```

Fig. 4 shows a simple transformation rule in Henshin taken from [5] for specifying an edit operation in MagicDraw [22]. The rule moves an existing Region from an existing State (the old source) to another existing State (the new source). This is done by deleting the containment edge region from the old source and recreating it in the new source. Rules specifying such edit operations may be used, e.g., to recognize semantic change sets while comparing two model versions [23,5].

The validity of basic constraints should be preserved throughout editing because a typical model editor is not able to display an instance violating them. Since FinalState is a subtype of State, applying the rule moveRegionFromStateToState might introduce a violation of the constraint no_region. Using OCL2AC [20], a language engineer can automatically integrate a constraint as an application condition into the rule and calculate the according constraint-guaranteeing version of the rule. The constraint-guaranteeing application condition obtained by integrating constraint no_region into rule moveRegionFromStateToState forbids matching this rule to a FinalState. It checks additionally if the model already encompasses a FinalState containing a Region – either matched by the rule or not. Fig. 5 presents the resulting constraint-guaranteeing application condition which is composed of 7 graphs (explained later in Example 6). Knowing the input model to be valid, most of the checks are unnecessary. Especially the checks which do not only involve elements being local to the rule application but amount to traversing every existing node, i.e., the global checks.

In this paper, we develop and implement optimizations that allow for omitting certain parts of a constraint-guaranteeing application condition during their construction. In our example, we will arrive at the optimized application condition shown in Fig. 6 which consists of only one graph that, moreover, requires a local check only. It forbids the rule node newSource:State to be matched to a FinalState. In Sect. 4.1, we will show that this simple application condition is con-

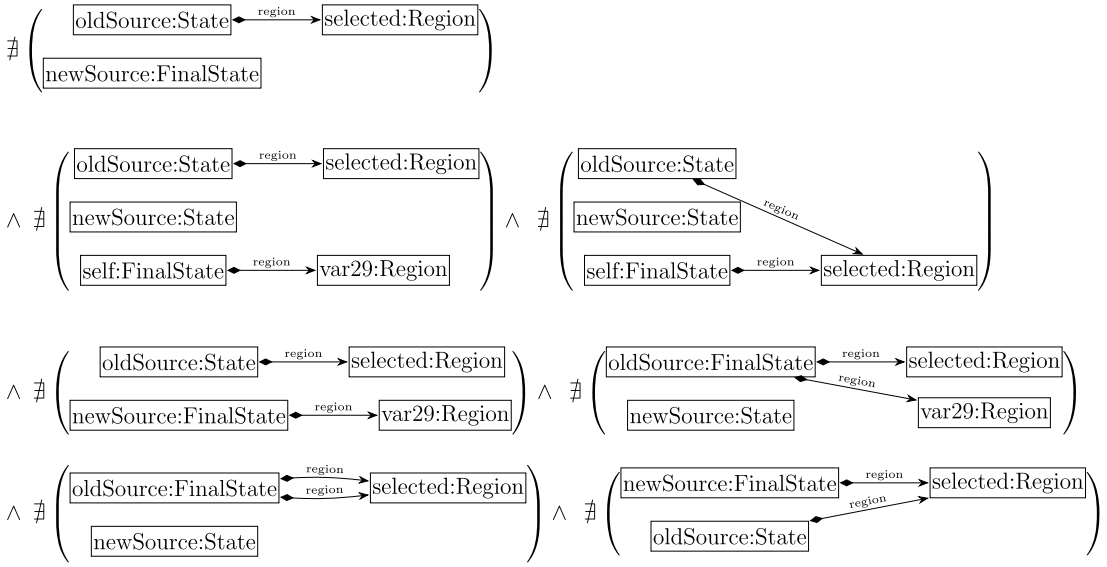


Fig. 5. Non-optimized constraint-guaranteeing application condition for moveRegionFromStateToState after integrating the constraint no_region.

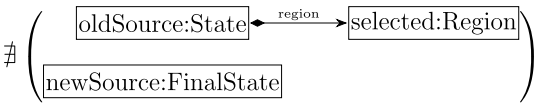


Fig. 6. Optimized application condition for moveRegionFromStateToState relative to the constraint no_region.

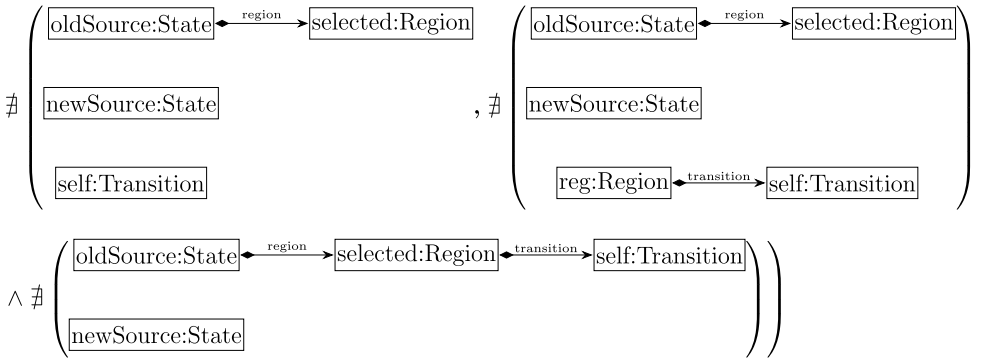


Fig. 7. Non-optimized constraint-guaranteeing application condition for moveRegionFromStateToState after integrating the constraint TransitionInRegion. The optimized one is just true.

straint preserving (even sustaining) and in Sect. 4.3 we will (formally) compare this with the complex constraint-preserving application condition resulting from the standard construction (see Fig. 15).

As the rule moveRegionFromStateToState does not change any graph element occurring in constraint TransitionInRegion, this constraint cannot be violated by a result model if it was not violated before. Hence, the optimized application condition is just true. Fig. 7 illustrates the corresponding constraint-guaranteeing application condition (the non-optimized one) as computed by OCL2AC. It consists of three graphs and states that there exists no Transition-node outside of the rule's match such that there exists no Region-node that contains the Transition-node; the Region-node might either stem from the match or not.[1]

Consequently, assuming valid input models, constraint-guaranteeing application conditions can be considerably simplified.

3. Formal background

Our approach is based on the theory of algebraic graph transformation [9,10]. EMF models and model transformations can be formalized as typed attributed graphs and graph transformations as presented in [11]. These graphs with adequate morphisms form, in turn, a so-called $\mathcal{M}$-adhesive category [18,10]. We state our results in this formal framework to also cover various graph-like and other model structures such as Petri nets. In the following, we recall (i) $\mathcal{M}$-adhesive categories and the category of typed graphs with inheritance as such a category, (ii) *nested constraints* and *conditions* as a means to express properties of objects and morphisms, and (iii) transformation rules which, altogether, set our formal background. Besides, we explain known techniques and results to integrate constraints as application conditions into transformation rules. In this paper, for simplicity, we restrict our presentation to the case of typed graphs with inheritance and (largely) neglect attribution. However, our general approach supports attributes as well because all our simplifications are proven to be correct for $\mathcal{M}$-adhesive categories with $\mathcal{E}'$-$\mathcal{M}$ pair factorization; the category of typed attributed graphs with inheritance is an instance [9].

3.1. Typed graphs with inheritance

We first introduce typed graphs with inheritance [9]. All abstract concepts that are introduced later on will be illustrated using such graphs as example. The definition of typed graphs is based on the definition of graphs and graph morphisms.

Definition 1 *(Graphs and graph morphisms).* A *graph* $G = (V_G, E_G, scr_G, tar_G)$ consists of two sets V_G and E_G, the nodes and edges of G, and two functions $scr_G, tar_G : E_G \to V_G$, its *source* and *target function*.

A *graph morphism* $f : G \to H$ between two graphs $G = (V_G, E_G, scr_G, tar_G)$ and $H = (V_H, E_H, scr_H, tar_H)$ is a pair of functions $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ between the node and the edge sets that is compatible with the source and

[1] This is of course equivalent to stating that every Transition-node is contained in a Region (either matched or not). However, for technical reasons, we only use the existential quantifier in our computed output.

target functions, i.e., $scr_H \circ f_E = f_V \circ scr_G$ and accordingly $tar_H \circ f_E = f_V \circ tar_G$. A graph morphism $f = (f_V, f_E)$ is *injective* (*surjective*) if both f_V and f_E are. Composition of graph morphisms is defined componentwise.

Definition 2 *(Typed graphs with inheritance. Clan morphism).* Given a graph TG, called *type graph*, a *graph typed over TG* is a tuple $(G, type_G)$ where G is a graph and $type_G : G \to TG$ is a graph morphism. A *typed graph morphism* f between two graphs $(G, type_G)$ and $(H, type_H)$, typed over the same type graph TG, is a graph morphism from G to H that respects the typing, i.e., $type_H \circ f = type_G$.

A *type graph with inheritance* $TGI = (TG, I, A)$ consists of a type graph $TG = (V_{TG}, E_{TG}, scr_{TG}, tar_{TG})$, a partial order $I \subset V_{TG} \times V_{TG}$, called *inheritance hierarchy*, and a set $A \subset V_{TG}$, called *abstract nodes*. The (*type*) *clan* of a node $n \in V_{TG}$ is defined by the set

$$clan_I(n) := \{m \in V_{TG} \mid (m, n) \in I\} \ .$$

We will abbreviate $(m, n) \in I$ as $m \leq n$.

A graph typed over TGI is a tuple $(G, type_G)$ where G is a graph and $type_G = (type_{GV}, type_{GE}) : G \to TG$ is a graph morphism such that

$$type_{GV} \circ scr_G(e) \leq scr_{TG} \circ type_{GE}(e)$$

and

$$type_{GV} \circ tar_G(e) \leq tar_{TG} \circ type_{GE}(e)$$

hold for all $e \in E_G$. Given a graph $(G, type_G)$ typed over TGI, $type_G = (type_{GV}, type_{GE})$ is called *clan morphism*, and G is called *concretely typed* if the preimage of A under $type_{GV}$ is empty, i.e., if no node of G is mapped to an abstract node in TG.

A *typed graph morphism* $f = (f_V, f_E)$ between two graphs $(G, type_G)$ and $(H, type_H)$ typed over the same type graph with inheritance TGI is a graph morphism from G to H such that additionally

$$type_{HV} \circ f_V(n) \leq type_{GV}(n) \tag{1}$$

for all $n \in V_G$ and

$$type_{HE} \circ f_E(e) = type_{GE}(e)$$

for all $e \in E_G$. Morphism f is *type refining* if there is a node $n \in V_G$ with $type_{HV} \circ f_V(n) < type_{GV}(n)$, otherwise it is called *type-strict* or *strong*.

Example 1. The statechart meta-model from our running example (Fig. 1) can be understood as a type graph with inheritance. Vertex is the only abstract node. The inheritance hierarchy is given by the special edges with unfilled arrowhead. This means, both Pseudostate and State inherit from the abstract node Vertex and FinalState inherits from State. By transitive closure, FinalState also inherits from Vertex. A graph is concretely typed over this type graph if it does not contain a node of type Vertex.

The constraints, rules, and application conditions displayed in Sect. 2 all contain graphs which are typed over that type graph. Instead of explicitly showing the clan morphism, we annotate the elements with the names of the nodes or edges to which they are mapped. We will explain typed graph morphism in a later example.

3.2. $\mathcal{M}$-adhesive categories

We now introduce the notion of an $\mathcal{M}$-*adhesive category* [10]. These categories generalize *adhesive categories* [24] which can be understood as axiomatizing categories where pushouts along monomorphisms behave like pushouts along injective functions in the category **Set**. In our presentation, we assume basic knowledge of category theory; an introduction that aims at graph transformation can be found in the appendix of [9].

Definition 3 *($\mathcal{M}$-adhesive category).* A category $\mathcal{C}$ with a class of monomorphisms $\mathcal{M}$ is $\mathcal{M}$-*adhesive* if

- the class $\mathcal{M}$ contains all isomorphisms and is closed under composition and decomposition of morphisms, i.e., whenever defined, $f, g \in \mathcal{M} \Rightarrow f \circ g \in \mathcal{M}$ and $f \circ g, f \in \mathcal{M} \Rightarrow g \in \mathcal{M}$;
- the category $\mathcal{C}$ has pullbacks along $\mathcal{M}$-morphisms and $\mathcal{M}$ is closed under such, i.e., wherever a square like the one depicted in Fig. 8 is a pullback along an $\mathcal{M}$-morphism n, also $m \in \mathcal{M}$;
- the category $\mathcal{C}$ has pushouts along $\mathcal{M}$-morphisms and $\mathcal{M}$ is closed under such; moreover, pushouts along $\mathcal{M}$-morphisms are *vertical weak van Kampen squares*. Here, a pushout square as depicted in Fig. 8 is a vertical weak van Kampen square if for every commutative cube over it (as depicted in Fig. 9) where the faces to the back are pullbacks and $a, b, c, d \in \mathcal{M}$, the top square is a pushout if and only if both front faces are pullbacks.

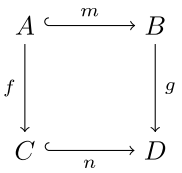


Fig. 8. A pushout along the $\mathcal{M}$-morphism m.

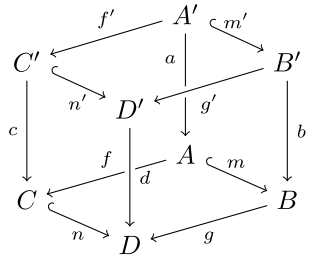


Fig. 9. Commutative cube over the pushout square from left.

For some of the following results to hold, we will need $\mathcal{M}$-adhesive categories with a further property, namely a factorization property for pairs of morphisms. This property is needed to prove the correctness of the Shift Construction (see Construction 1 and Fact 1 below).

Definition 4 *($\mathcal{E}'$-$\mathcal{M}$ pair factorization).* Given an $\mathcal{M}$-adhesive category $\mathcal{C}$ with a class of monomorphisms $\mathcal{M}$ and a class $\mathcal{E}'$ of pairs of morphisms with the same codomain, the category $\mathcal{C}$ has a *unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization* if, for each pair of morphisms $f_1 : A_1 \to C, f_2 : A_2 \to C$ with the same codomain, there exists a unique (up to isomorphism) object K, unique (up to isomorphism) morphisms $e_1 : A_1 \to K, e_2 : A_2 \to K$ and $m : K \to C$ with $(e_1, e_2) \in \mathcal{E}'$ and $m \in \mathcal{M}$ such that $m \circ e_1 = f_1$ and $m \circ e_2 = f_2$.

Provided that the inheritance structure has a largest object (which is no restriction in practice as one can always add one), graphs typed over a fixed type graph with inheritance have been shown to constitute an $\mathcal{M}$-adhesive category (and even an *adhesive HLR* category which is a stronger notion of adhesiveness) with respect to the class of monomorphisms $\mathcal{M}$ that contains all strong injective morphisms [25]. (Note that in such categories of graphs, monotonicity coincides with a morphism being injective.) It is not difficult to check that this category has an $\mathcal{E}'$-$\mathcal{M}$ pair factorization of morphisms: Each pair of morphisms with the same codomain might be (uniquely) factored into a pair of jointly surjective morphisms followed by an injective strong one. Alternatively, type graphs with inheritance may be *flattened* into equivalent normal type graphs; the category of typed graphs is also known to be $\mathcal{M}$-adhesive and possess an $\mathcal{E}'$-$\mathcal{M}$ pair factorization of morphisms [9]. In any case, all results abstractly formulated for $\mathcal{M}$-adhesive categories in the following are valid for typed graphs with inheritance in particular. Further examples include sets and functions, typed attributed graphs, algebraic signatures and signature morphisms, or elementary Petri nets [9].

3.3. Nested conditions and constraints

Given an $\mathcal{M}$-adhesive category, *nested constraints* express properties of its objects whereas *nested conditions* express properties of morphisms [13]. Conditions are mainly used to restrict the applicability of rules. Constraints are special kinds of conditions; both are defined recursively as trees of morphisms. For the definition of constraints, we assume the existence of an initial object $\emptyset$ in the category $\mathcal{C}$ (which is just the empty graph in the case of typed graphs with inheritance).

Definition 5 *((Nested) conditions and constraints).* Let $\mathcal{C}$ be an $\mathcal{M}$-adhesive category with initial object $\emptyset$. Given an object P, a *(nested) condition* over P is defined recursively as follows: $\texttt{true}$ is a condition over P. If $a : P \to C$ is a morphism and d is a condition over C, $\exists(a : P \to C, d)$ is a condition over P again. Moreover, Boolean combinations of conditions over P are conditions over P. A *(nested) constraint* is a condition over the initial object $\emptyset$. We call constraints of the form $\exists(a : \emptyset \to C, \texttt{true})$ *positive* and those of the form $\neg\exists(a : \emptyset \to C, \texttt{true})$ *negative*. We say that an object C *occurs in* a condition c, denoted as $C \in c$, if c is defined as condition over C or C is the codomain of one of the morphisms that constitute c.

Satisfaction of a nested condition c over P for a morphism $g : P \to G$, denoted as $g \models c$, is defined as follows: Every morphism satisfies $\texttt{true}$. The morphism g satisfies a condition of the form $c = \exists(a : P \to C, d)$ if there exists an $\mathcal{M}$-morphism $q : C \hookrightarrow G$ such that $g = q \circ a$ and $q \models d$. For Boolean operators, satisfaction is defined as usual. An object G satisfies a constraint c, denoted as $G \models c$, if the empty morphism to G does so. A condition c_1 over P *implies* a condition

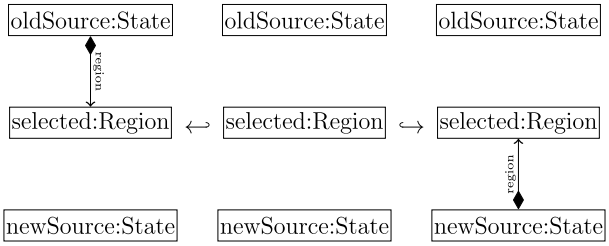


Fig. 10. The graphical formal representation of the Henshin rule moveRegionFromStateToState depicted in Fig. 4.

c_2 over P, denoted as $c_1 \Rightarrow c_2$, if for every morphism $g : P \to G$ with $g \models c_1$ also $g \models c_2$. Two conditions are equivalent, denoted as $c_1 \equiv c_2$, when $c_1 \Rightarrow c_2$ and $c_2 \Rightarrow c_1$. Implication and equivalence for constraints is obtained by the definition for conditions.

In the case of graphs, the graph constraints resulting from the above definition are expressively equivalent to a first-order logic on graphs [13,12]. To ease notation, we drop the domain of morphisms in conditions whenever they may be unambiguously inferred and indicate the mapping by the names of nodes in figures; we also drop the `true` at the end of conditions. For example, we abbreviate $\exists(a_1 : \emptyset \to C_1, \neg\exists(a_2 : C_1 \to C_2, \texttt{true}))$ as $\exists(C_1, \neg\exists C_2)$ whenever it is clear that this denotes a constraint. Moreover, we introduce the abbreviations $\texttt{false} := \neg\texttt{true}$ and $\forall(C, d) := \neg\exists(C, \neg d)$ where d is some condition over C. Examples for graph constraints and conditions with informal explanation of their semantics are given in Sect. 2; all examples use the simplified notation.

3.4. Rule-based transformations

Rules are a technical means to declaratively define model transformations.

Definition 6 *(Rules and transformations).* A *rule* $\rho = (p, lac, rac)$ consists of a *plain rule* p and *left* and *right application conditions* *lac* and *rac*. The plain rule p consists of three objects L, K, and R, called *left-hand side* (LHS), *interface*, and *right-hand side* (RHS) with two $\mathcal{M}$-morphisms $l : K \hookrightarrow L, r : K \hookrightarrow R$. The application conditions *lac* and *rac* are nested conditions over L and R, respectively. The inverse rule ρ^{-1} of a rule $\rho = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R, lac, rac)$ is the rule $(R \overset{r}{\hookleftarrow} K \overset{l}{\hookrightarrow} L, rac, lac)$. A plain rule p is *monotonic* if $l : K \hookrightarrow L$ is an isomorphism, and *only deletes* if $r : K \hookrightarrow R$ is an isomorphism.

Given a rule $\rho = ((L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R), lac, rac)$ and a morphism $m : L \to G$, a *(direct) transformation* $G \Rightarrow_{\rho,m} H$ from G to H is given by the diagram to the right where both squares are pushouts, $m \vDash lac$, and $n \vDash rac$. If such a transformation exists, the morphism m is called a *match* and rule ρ is *applicable* at match m; in this case, n is called the *co-match* of the transformation. By $[\![\rho]\!]$ we denote the class of pairs of objects (G, H) such that there is a transformation $G \Rightarrow_{\rho,m} H$ for some match m.

$$\begin{array}{ccccc} L & \overset{l}{\hookleftarrow} & K & \overset{r}{\hookrightarrow} & R \\ {\scriptstyle m \vDash lac}\downarrow & & \downarrow{\scriptstyle k} & & \downarrow{\scriptstyle n \vDash rac} \\ G & \underset{e}{\hookleftarrow} & D & \underset{f}{\hookrightarrow} & H \end{array}$$

In $\mathcal{M}$-adhesive categories, the second pushout can always be constructed if the first one can, i.e., if $m \circ l$ has a *pushout complement* D. In the case of graph-like structures, applying a rule at a match m in G that fulfills the left application condition, the elements of $m(L \setminus K)$ are deleted. Then, at the chosen image of K in G, a copy of $R \setminus K$ is created. Afterwards, the resulting mapping of R into the new graph is checked to fulfill the right application condition of the rule. If it does, the new graph is the result of the rule application. Note that in the case of graph-like structures, the first pushout square exists if and only if the match m fulfills the *dangling edge* and the *identification conditions*. These conditions ensure that a rule application at this match would not let a context edge dangle, that deleted elements are only deleted once, and there is no conflict between deleting and preserving an element [9]. Moreover, in the case of typed graphs with inheritance, rules are expected to not create nodes of abstract types, i.e., $type_{RV}(V_R \setminus V_K) \cap A = \emptyset$.

Example 2. An example of a rule is shown in Fig. 4 in the style of Henshin where all graphs of the rule are integrated into a single graph and its elements are annotated according to their roles. Its semantics was already explained. Fig. 10 illustrates how this rule accords to the formal definition from above where the left column displays the LHS, the right column the RHS, and the central column the interface. These three graphs are connected by typed graph morphisms from the interface to the LHS and the RHS. A node is mapped to the node with the same name; e.g., the node of type State with the name newSource from the interface is mapped to the according nodes in the LHS and RHS.

Note that the restriction to strong morphisms in the rule forbids retyping by rule application. All elements are mapped to elements with exactly the same type (this analogously holds if one uses the flattening of type hierarchies). When applying the rule moveRegionFromStateToState, the States may be matched to FinalStates in an instance graph since FinalState inherits from State. But inside a rule, a State-node of the interface graph must not be mapped to a FinalState-node in the LHS or RHS.

3.5. Guaranteeing and preserving constraints

We are interested in ensuring that a given constraint c holds after rule applications. In several application scenarios it is enough to ensure this for cases where it is known that the input model of a transformation already satisfies the constraint. We first formally define the relevant notions before recalling the standard approach [13].

Definition 7 *(Guarantee and preservation of constraints).* Given a constraint c, a rule ρ is *c-guaranteeing* if, for every direct transformation $G \Rightarrow_\rho H$, the result satisfies c, i.e., $H \models c$. A rule ρ is *c-preserving* if, for every direct transformation $G \Rightarrow_\rho H$ where G is known to satisfy c, the result also satisfies c, i.e., $G \models c \Rightarrow H \models c$.

If we do not want to stress the constraint c or the constraint is clear from the context, we speak of *constraint-guaranteeing* or *-preserving rules*.

A recently introduced notion is the one of *consistency sustainment* [16]. It formalizes the intuition that the application of a rule does not introduce new violations of a given constraint. The form of consistency sustainment we use in this paper, has actually been introduced as *direct consistency sustainment* in [16]. There, *consistency sustainment* requires the number of violations to not grow; in particular, a transformation can be consistency sustaining even though it introduces new violations as long as it repairs enough existing ones. In contrast, *direct consistency sustainment* completely forbids the introduction of new violations. However, as this is the only kind of consistency sustainment we consider, we drop "direct" to keep terminology simpler. We only recall this definition for *negative constraints* as we will only need it in this specific context.

Definition 8 *(Consistency sustainment).* Let a negative constraint $c = \neg\exists\, C$ and a rule ρ be given. A transformation $G \Rightarrow_{\rho,m} H = G \overset{e}{\hookleftarrow} D \overset{f}{\hookrightarrow} H$ via rule ρ at match m is *consistency sustaining w.r.t. c* if for every $\mathcal{M}$-morphism $q' : C \hookrightarrow H$ there is a morphism $q_D : C \to D$ such that $f \circ q_D = q'$. Rule ρ is *consistency sustaining w.r.t. c* if all transformations from $[\![\rho]\!]$ are.

A transformation (rule) that is consistency sustaining w.r.t. a given constraint is preserving w.r.t. this constraint [16, Theorem 1]. The differences between the notions can hence be summarized as follows: Given a constraint c, every result of the application of a c-guaranteeing rule satisfies c (which corresponds to $\{\mathtt{true}\}\,\rho\,\{c\}$ in Hoare triple notation). A c-preserving rule only warrants this for already valid input (i.e., $\{c\}\,\rho\,\{c\}$ in Hoare triple notation). For input not satisfying c, the behavior of a c-preserving rule is not further restricted. This is refined by the definition of consistency sustainment: An application is not allowed to "increase" the amount of violation of c (in a certain technical sense) even on invalid input. We give concrete examples in Sect. 4.3.

While for special cases, constructions which turn out to produce consistency-sustaining application conditions can be found in the literature, e.g., [26], no general automatic construction has been presented, yet. A well-known approach to the construction of constraint-preserving and -guaranteeing rules is the construction of suitable application conditions for a given rule (without altering the actions of a rule). This construction is formally based on the following *shift-constructions* which allow "moving" conditions along morphisms or rules. The first step assumes a class $\mathcal{E}'$ of pairs of morphisms with the same codomain to be given; for typed graphs with inheritance, as in many application cases, this is the class of pairs of jointly surjective morphisms. The intuitive idea behind this construction is to iteratively overlap all objects that occur in the constraint (along the nesting structure) with the codomain of the morphism along which the condition is shifted, in all possible ways.

Construction 1 *(Shift along morphism).* Given a condition c over an object P and a morphism $b : P \to P'$, the *shift of c along b*, denoted as $\mathrm{Shift}(b, c)$, is inductively defined as follows:

- If $c = \mathtt{true}$,

$$\mathrm{Shift}(b, c) := \mathtt{true}\ .$$

- If $c = \exists\,(a : P \to C, d)$,

$$\mathrm{Shift}(b, c) := \bigvee_{(a',b') \in \mathcal{F}} \exists\,(a', \mathrm{Shift}(b', d))\ ,$$

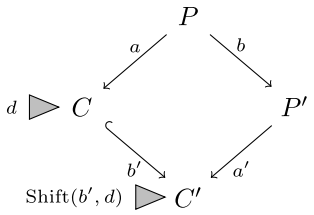


Fig. 11. Graphical representation of the definition of $\mathrm{Shift}(b, \exists\,(a : P \to C, d))$.

where $\mathcal{F}$ contains morphism pairs from $\mathcal{E}'$ with one morphism from $\mathcal{M}$ that complement a and b to a commutative diagram, i.e.,

$$\mathcal{F} := \{(a', b') \in \mathcal{E}' \mid b' \in \mathcal{M} \text{ and } b' \circ a = a' \circ b\} \ .$$

This definition is illustrated in Fig. 11 for one of such pairs of morphisms. Note that the empty disjunction is equivalent to `false`.

- If $c = \neg d$,

$$\mathrm{Shift}(b, c) := \neg\,\mathrm{Shift}(b, d) \ .$$

- If $c = \bigwedge_{i \in I} c_i$,

$$\mathrm{Shift}(b, c) := \bigwedge_{i \in I} \mathrm{Shift}(b, c_i) \ .$$

Remark 1. The Shift construction presented above is the one needed in the general case of matching for which we present our theory. In case one restricts rule application to matches $m \in \mathcal{M}$, a subset of the pairs of morphisms $\mathcal{F}$ is enough: One only needs to consider those pairs of morphisms (a', b'), where both $a', b' \in \mathcal{M}$; see, e.g., [15]. This is also the construction upon which our implementation is based, i.e., our implementation supports injective matching of graph transformation rules. Moreover, as we will discuss in more detail later, Henshin allows for type refinement in the check of application conditions, which means that we also allow for type refinement in both a' and b'. Accordingly, in our implementation, the class $\mathcal{F}$ contains pairs of morphisms which are jointly surjective and both injective, and where none of the morphisms is required to be type-strict.

Fact 1 *(Correctness of* Shift *[27, Lemma 3.11]). In an $\mathcal{M}$-adhesive category $\mathcal{C}$ with $\mathcal{E}'$-$\mathcal{M}$ pair factorization, given a nested condition c over an object P and a morphism $b : P \to P'$, for each morphism $g' : P' \to G$*

$$g' \models \mathrm{Shift}(b, c) \Leftrightarrow g := g' \circ b \models c \ .$$

As an important consequence of this lemma, in categories with initial object the shift of conditions along morphisms can be used to integrate a given constraint as right application condition into a transformation rule. The resulting right application condition is satisfied by the co-match of a rule application if and only if the result of the transformation satisfies the constraint: Constraints are defined over the initial object and thus can be shifted along the unique morphism from the initial object to the RHS of the rule to become a right application condition. The next example illustrates this for the case of injective matching.

Example 3. To shift the constraint no_region along the morphism from the empty graph to the RHS of the rule moveRegionFromStateToState, we need to overlap its RHS with the only graph of the constraint in every possible way. Fig. 12 shows the resulting right application condition. It consists of 7 graphs and is formed in the following way: The first graph (the uppermost graph) results from a maximal overlapping of the constraint with the rule, i.e., by identifying as many elements as possible. Note that it is possible to identify nodes of type State from the rule with the node of type FinalState from the constraint since FinalState is a subtype of State (compare Fig. 1). The second graph results from copying the graph of the constraint and the RHS of the rule and putting them next to each other. The third graph results from merging the nodes of type Region. The fourth and the fifth graph result from just merging nodes of type State and FinalState. The sixth and the seventh graph result from merging the nodes of type State and the nodes of type Region.

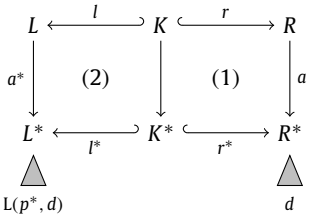
Similarly, conditions can be "moved" along rules.

Construction 2 *(Shift over rule).* Given a condition c over an object R and a plain rule $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$, the *shift of c over p*, denoted as $\mathrm{L}(p, c)$, is inductively defined as follows:

- If $c = \texttt{true}$,

$$\mathrm{L}(p, c) \coloneqq \texttt{true} \ .$$

- If $c = \exists\,(a : R \to R^*, d)$, consider the following diagram:



If $a \circ r$ has a pushout complement (1) and

$$p^* = (L^* \overset{l^*}{\hookleftarrow} K^* \overset{r^*}{\hookrightarrow} R^*)$$

is the rule derived by constructing the pushout (2), i.e., by applying the inverse rule p^{-1} at match a to R^*, then

$$\mathrm{L}(p, c) \coloneqq \exists\,(a^* : L \to L^*, \mathrm{L}(p^*, d)) \ .$$

Otherwise, i.e., if the pushout complement (1) does not exist,

$$\mathrm{L}(p, c) \coloneqq \texttt{false} \ .$$

- If $c = \neg d$,

$$\mathrm{L}(p, c) \coloneqq \neg\,\mathrm{L}(p, d) \ .$$

- If $c = \bigwedge_{i \in I} c_i$,

$$\mathrm{L}(p, c) \coloneqq \bigwedge_{i \in I} \mathrm{L}(p, c_i) \ .$$

Fact 2 *(Correctness of* L *[27, Lemma 3.14]). In an $\mathcal{M}$-adhesive category $\mathcal{C}$, given a nested condition c over an object R and a plain rule $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$, for each direct transformation $G \Rightarrow_{p,m,m^*} H$ where m^* denotes the co-match of that transformation,*

$$m \models \mathrm{L}(p, c) \Leftrightarrow m^* \models c \ .$$

In particular, the construction above allows one to equivalently shift all right application conditions of a rule over this rule to its LHS; hence, we can safely assume that rules only have left application conditions which we denote as ac. More importantly, the combination of both constructions allows to integrate constraints as application conditions into rules such that they guarantee or preserve the validity of the constraint. Starting with special cases in the category of graphs in [28], this approach has been generalized to arbitrary nested constraints in the general setting of $\mathcal{M}$-adhesive categories [13].

Construction 3 *(Constraint-guaranteeing and -preserving application conditions).* In an $\mathcal{M}$-adhesive category $\mathcal{C}$ with initial object $\emptyset$ and $\mathcal{E}'$-$\mathcal{M}$ pair factorization, given a nested constraint c and a rule $\rho = (p, ac)$ with plain rule $p = (L \hookleftarrow K \hookrightarrow R)$ and left application condition ac, the c-guaranteeing version of ρ, denoted as $\mathrm{Gua}(\rho, c)$, is defined as follows:

$$\mathrm{Gua}(\rho, c) \coloneqq (p, ac \wedge ac_{\mathrm{gua}}) \ ,$$

where

$$ac_{\mathrm{gua}} \coloneqq \mathrm{L}(p, \mathrm{Shift}(\emptyset \to R, c)) \ .$$

Similarly, the c-preserving version of ρ, denoted as $\mathrm{Pres}(\rho, c)$, is defined as follows:

$$\mathrm{Pres}(\rho, c) \coloneqq (p, ac \wedge ac_{\mathrm{pres}}) \ ,$$

where

$$ac_{\mathrm{pres}} \coloneqq \mathrm{Shift}(\emptyset \to L, c) \Rightarrow ac_{\mathrm{gua}} \ .$$

Intuitively, for ac_{gua} one first computes all possible ways in which a constraint can be valid after rule application using shift along a morphism. The possibilities are then shifted over the rule, resulting in an application condition that requires one of these possibilities to be the case. When computing ac_{pres}, one adds the assertion that the input is already valid as a precondition to ac_{gua}.

Fact 3 *(Correctness of* Pres *and* Gua *[13, Corollary 5]). Let $\mathcal{C}$ be an $\mathcal{M}$-adhesive category with initial object and $\mathcal{E}'$-$\mathcal{M}$ pair factorization for a class $\mathcal{E}'$ of pairs of morphisms with the same codomain. Then, for every rule ρ and every constraint c,*

- $[\![\text{Gua}(\rho, c)]\!] \subseteq [\![\rho]\!]$ *and* $\text{Gua}(\rho, c)$ *is c-guaranteeing, and*
- $[\![\text{Pres}(\rho, c)]\!] \subseteq [\![\rho]\!]$ *and* $\text{Pres}(\rho, c)$ *is c-preserving.*

Example 4. Shifting the right application condition computed in Example 3 (see Fig. 12) over the rule moveRegionFromStateToState results in the left application depicted in Fig. 5: The inverse rule of moveRegionFromStateToState is applicable

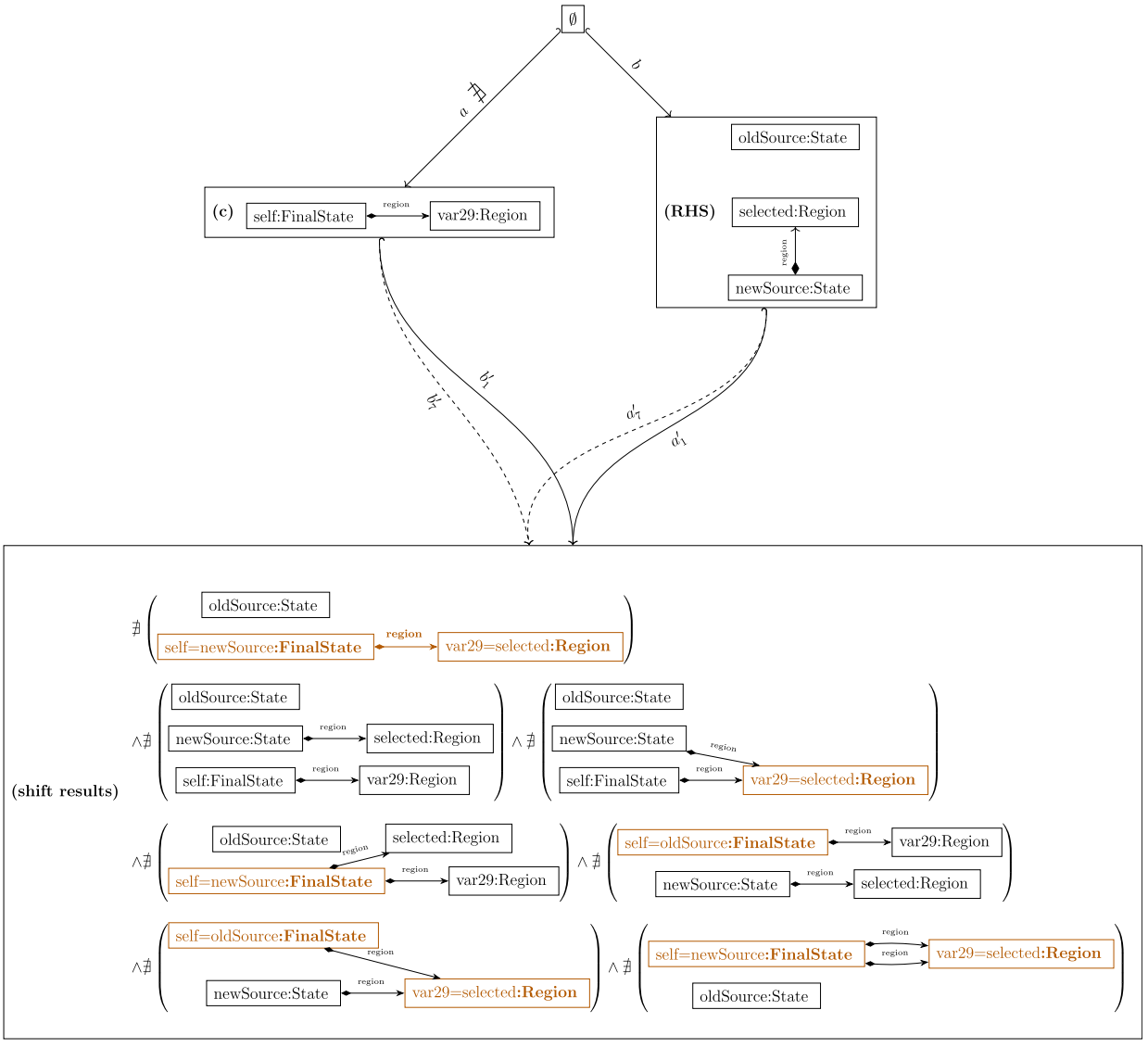


Fig. 12. The result of shifting the constraint no_region shown in Fig. 3 along the corresponding empty morphism to the RHS of the rule moveRegionFromStateToState depicted in Fig. 4. The overlapped elements are colored brown and if two nodes are merged, their names are concatenated together using "=". Furthermore, De Morgan's law has been applied to replace the disjunction by a conjunction. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

to all graphs occurring in the right application condition and moves the region-edge that stems from the RHS of the rule to the second node of type State in the rule.

The overall result of Gua(moveRegionFromStateToState,no_region), i.e., the constraint-guaranteeing rule as defined in Construction 3, is the rule moveRegionFromStateToState equipped with that application condition. The resulting application condition of the constraint-preserving rule can be found in Fig. 15 and is discussed later in Sect. 4.3.

Fig. 7 shows the resulting application condition when computing the constraint-guaranteeing version of this rule for the constraint TransitionInRegion.

4. Optimizing application conditions

The constructions Gua and Pres, as recalled in the last section, allow constructing a constraint-guaranteeing or -preserving rule without altering its actions. This raises further questions: First, from a logical point of view, the application conditions used for that should be as weak as possible to allow for as many applications of the equipped rule as possible. Application condition `false`, for example, trivially is constraint-guaranteeing and -preserving for any rule relative to any constraint but often not the weakest condition. Secondly, the resulting application conditions should be easy to compute and, to be understandable, "simple" from a structural point of view.

From a logical point of view, the application conditions ac_{gua} and ac_{pres} computed in Construction 3 are as weak as possible to serve their respective purposes; see [13, Sect. 7] and also Fact 4 in Sect. 4.3. In practical applications, however, the construction $\text{Gua}(\rho, c)$ quite frequently leads to unnecessary restrictive application conditions when the input of a transformation is already known to satisfy a given constraint. While logically weaker, without further and costly simplifications, the application conditions arising by $\text{Pres}(\rho, c)$ are rather complex from a structural point of view and their computation extends the computation of $\text{Gua}(\rho, c)$. Moreover, they have the property that they do not further control the application on invalid input (see [16] and Sect. 4.3). This can also be undesired, depending on the application scenario.

To overcome these issues, we define the notions of *weakest constraint-preserving* and *weakest consistency-sustaining application conditions* in this section and show how, in some cases, these can be directly computed during the computation of ac_{gua} by omitting certain parts. The general idea is to check whether a rule can introduce violations of a constraint at all before integrating this constraint as application condition. In special cases, if integration is necessary, at least those parts of ac_{gua} that just check for antecedent validity of the constraint can be dropped. This optimization is defined in the general context of $\mathcal{M}$-adhesive categories and subsequently compared formally to the computation of ac_{pres} as defined in [13]. Besides, we present further simplifications of application conditions that are specific to constraints imposed by the EMF framework.

In the case of typed graphs, our simplifications formalize the following intuitive ideas:

1. We collect all rule elements being deleted or created and check if this set shares types with the constraint elements, respecting the clans of the node types. If no types are shared, an application of this rule cannot affect the validity of the constraint in any way. Hence, the resulting application condition is just `true`.
2. If a rule creates new graph structure only, positive constraints $\exists C$ do not need to be integrated into such a rule. Analogously, if a rule only deletes graph structure, negative constraints $\neg\exists C$ do not need to be integrated. In both cases, applications of such a rule cannot introduce a new violation of the constraint. Hence, the resulting application condition is just `true`.
3. While calculating ac_{gua}, the graphs of the constraint are overlapped with the RHS of the rule in all possible ways. For negative constraints $\neg\exists C$ it is not necessary to consider all possible overlappings. One may omit all the cases where C and the RHS R do not overlap in at least one element created. The parts of ac_{gua} arising from these cases would just check that the input graph already fulfills the constraint.

These simplifications are stated formally and proven to result in weakest constraint-preserving or weakest consistency-sustaining application conditions in Theorem 1. The first simplification is completely general in scope whereas the second and the third are specific to a certain type of rule and/or constraint. Our experience shows, however, that negative constraints are the ones that are used most frequently. Also, rules that only delete or only create are commonly used. For example, so-called *triple graph grammars*, an established mechanism for bidirectional transformation, are defined using only monotonic rules [29]. This means that also the last two simplifications can contribute to a valuable optimization, especially when integrating several given constraints as application conditions into each rule of a larger set. Moreover, especially the third simplification omits cases where an arising graph in an application condition contains nodes which are not connected to nodes of the LHS of the rule. Thus, this simplification prevents a number of global checks in application conditions.

A second set of simplifications omits parts of application conditions that would require patterns to exist which are known to be always false for EMF models (like parallel edges of the same type between the same two nodes). These simplifications are formally stated and proven to not alter the semantics of an application condition in the context of EMF models in Theorem 2.

Based on both sets of simplifications, we developed and implemented an approach to computing application conditions that is illustrated in Fig. 13: If a constraint is to be integrated as application condition into a rule, we first check if this is necessary at all (Simplifications 1. and 2. from above). If it is, we start the computation of $\text{Gua}(\rho, c)$ as defined in

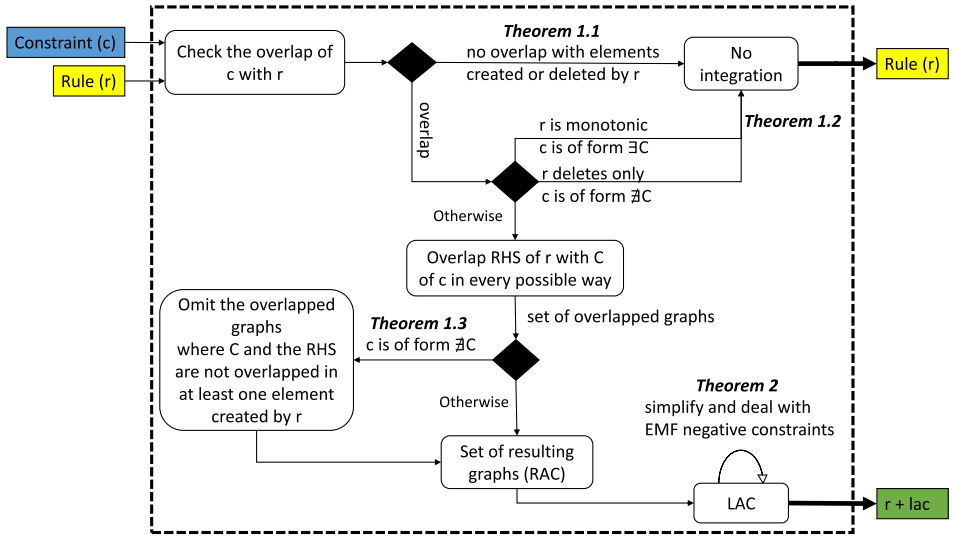


Fig. 13. Optimization process based on the theorems presented.

Construction 3. *During* that computation, we check at least if the constraint is negative such that we can drop parts that only check for antecedent validity of the constraint (Simplification 3. from above); these parts are then dropped from the right application condition that arises during the construction. The remaining right application condition is shifted over the rule to its LHS and the result is simplified according to the EMF specific simplifications. In case any of the Simplifications 1. – 3. took place, Theorem 1 and Corollary 3 imply that the result is either a weakest constraint-preserving or even a weakest consistency-sustaining application condition. Otherwise, Corollary 3 implies that the result is a weakest constraint-guaranteeing application condition.

4.1. Weakest forms of application conditions

We first define and characterize *weakest application conditions* in their preserving and sustaining variants. The idea is that of the logically weakest possible application condition on the class of matches such that a rule becomes preserving or sustaining, respectively.

Definition 9 *(Weakest forms of application conditions).* Let a plain rule $p = (L \hookleftarrow K \hookrightarrow R)$ and a constraint c be given. An application condition ac over L is a *c-preserving application condition* if the rule

$$\rho := (p, ac)$$

is c-preserving (in the sense of Definition 7), i.e., if for all objects G with $G \models c$ and all objects H and matches $m : L \to G$ with $m \models ac$ and $G \Rightarrow_{\rho,m} H$ also $H \models c$. A c-preserving application condition ac_{wp} for p is a *weakest c-preserving application condition* if, on the class of matches for p, it is implied by every c-preserving application condition ac for p, i.e., if for all c-preserving application conditions ac for all matches m for p, $m \models ac \Rightarrow m \models ac_{\text{wp}}$.

Analogously, ac is a *consistency-sustaining application condition w.r.t. c* if ρ is consistency sustaining (in the sense of Definition 8). A consistency-sustaining application condition ac_{ws} is a *weakest consistency-sustaining application condition* if, on the class of matches for p, it is implied by every consistency-sustaining application condition ac, i.e., if for all consistency-sustaining application conditions ac and for all matches m for p, $m \models ac \Rightarrow m \models ac_{\text{ws}}$.

As already done in the definition above, we speak of *(weakest) constraint-preserving* or *(weakest) consistency-sustaining application conditions* if we do not want to stress a specific constraint or the constraint is clear from the context.

Example 5. While `false` is a trivial constraint-preserving application condition for any rule relative to any constraint, Fig. 5 displays a no_region-preserving application condition for the rule moveRegionFromStateToState (Fig. 4); it is even no_region-guaranteeing. We will argue that the condition displayed in Fig. 6 is a weakest consistency-sustaining application condition for that rule w.r.t. the constraint no_region. Obviously, the first application condition implies the second as the second is a part of the conjunction that constitutes the first one.

We defined weakest application conditions for plain rules relative to a single constraint. This is enough since these constraint-preserving application conditions can be combined with each other and already existing ones via conjunction.

Lemma 1 *(Reduction to plain rules). Let a rule $\rho = (p, ac)$ with plain rule $p = (L \hookleftarrow K \hookrightarrow R)$ and constraints c_1 and c_2 be given. Let ac_{wp1} and ac_{wp2} be weakest constraint-preserving application conditions for the rule p relative to c_1 or c_2, respectively. Then*

1. *$ac_{wp1} \wedge ac_{wp2}$ is a weakest constraint-preserving application condition for the rule p relative to $c_1 \wedge c_2$, and*
2. *$ac \wedge ac_{wp1}$ is a logically weakest application condition that preserves c_1 such that $[\![(p, ac \wedge ac_{wp1})]\!] \subseteq [\![\rho]\!]$. This means, for any other c_1-preserving application condition ac'*

$$([\![(p, ac')]\!] \subseteq [\![\rho]\!]) \Rightarrow (ac' \Rightarrow ac_{wp1}) \ .$$

The analogous statements hold in the case of weakest consistency-sustaining application conditions.

Proof. We only prove the second statement concerning preservation as the proof of the first is similar but even simpler; proofs for the sustaining case are completely analogous.

Let ac' be an c_1-preserving application condition over L such that $[\![(p, ac')]\!] \subseteq [\![\rho]\!]$. Then, since both sets are defined by applications of the same plain rule p, $ac' \Rightarrow ac$. Moreover, since ac_{wp1} is a weakest c_1-preserving application condition for p and ac' is c_1-preserving, also $ac' \Rightarrow ac_{\text{wp1}}$. Hence, $ac' \Rightarrow (ac \wedge ac_{\text{wp1}})$. □

Depending on the desired application case, one might want to use such a weakest version of an application condition to replace an existing application condition (e.g., when the existing one was designed to serve the same purpose but is logically stronger or structurally more complex) or to combine both (e.g., when the existing one serves another purpose and the semantics of the rule should not be extended). However, the above lemma shows that it is enough to consider plain rules in the following.

Like for the weakest liberal preconditions introduced in [13], which are constraints instead of application conditions, there are simple characterizations for weakest constraint-preserving and weakest consistency-sustaining application conditions. Intuitively, for both kinds of application conditions, the weakest ones are those that do not block any sound (i.e., preserving or sustaining, respectively) transformations. We are going to use this characterization to compare our constructed consistency-sustaining rules with the preserving ones from Fact 3. The statement concerning sustainment is restricted to the case of negative constraints as we only introduced them for this specific case.

Lemma 2 *(Characterization of weakest application conditions). Let a constraint c and a plain rule p be given. A condition ac_{wp} over L is a weakest c-preserving application condition for p if (i) for all objects G with $G \models c$ and all morphisms $m : L \to G$*

$$\big(m \models ac_{wp}\big) \Leftrightarrow \big((G \Rightarrow_{m,p} H) \Rightarrow (H \models c)\big) \tag{2}$$

and (ii) for all objects G with $G \not\models c$ and all morphisms $m : L \to G$

$$G \Rightarrow_{m,p} H \Rightarrow m \models ac_{wp} \ . \tag{3}$$

Similarly, a condition ac_{ws} over L is a weakest consistency-sustaining application condition for p w.r.t. $c = \neg\exists C$ if for all objects G and all morphisms $m : L \to G$, $m \models ac_{ws}$ if and only if $G \Rightarrow_{m,p} H$ implies that for all $\mathcal{M}$-morphisms $q' : C \hookrightarrow H$ there exists an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ such that $f \circ q_D = q'$, i.e.,

$$m \models ac_{ws} \Leftrightarrow \big((G \Rightarrow_{m,p} H) \Rightarrow (\forall q' : C \hookrightarrow H \in \mathcal{M}. \exists q_D : C \hookrightarrow D \in \mathcal{M}. f \circ q_D = q')\big) \tag{4}$$

where $G \overset{e}{\hookleftarrow} D \overset{f}{\hookrightarrow} H$ denotes the respective transformation.

Proof. Both statements can be proved by suitably adapting the corresponding proof for weakest liberal preconditions in [13, Fact 8]. We start with the statement for preservation. By the first direction of Eq. (2), ac_{wp} is a c-preserving application condition for p. Let ac be another c-preserving application condition for p. Let $m : L \to G$ be an arbitrary morphism such that $m \models ac$ and $G \Rightarrow_{m,p} H$. First, assume $G \models c$. Since ac is a c-preserving application condition and $G \models c$, $G \Rightarrow_{m,p} H$ implies $H \models c$. Then, by the second direction of Eq. (2), $m \models ac_{\text{wp}}$. Secondly, assume $G \not\models c$. Then, by Eq. (3) and $G \Rightarrow_{m,p} H$, $m \models ac_{\text{wp}}$. Summarizing, $ac' \Rightarrow ac_{\text{wp}}$ and ac_{wp} is a weakest c-preserving application condition.

In the case of sustainment, again, the first direction of Eq. (4) ensures that ac_{ws} is a consistency-sustaining application condition w.r.t. c. Let a consistency sustaining application condition ac of p w.r.t. c be given. Then, for any transformation $G \Rightarrow_{m,p} H$ such that $m \models ac$, consistency sustainment of ac implies that the second statement of Eq. (4) is true. Hence, this equivalence implies $m \models ac_{\text{ws}}$ and, thus, $m \models ac \Rightarrow m \models ac_{\text{ws}}$ for matches m for p. □

As a simple corollary of the above, we obtain the semantic equivalence of weakest preserving and weakest sustaining application conditions for matches in valid objects:

Corollary 1 *(Semantic equivalence on valid objects). Given a negative constraint $c = \neg\exists\, C$ and a plain rule $p = (L \hookleftarrow K \hookrightarrow R)$, then, for every match $m : L \to G$ where $G \models c$, the equivalence*

$$m \models ac_{wp} \Leftrightarrow m \models ac_{ws} \tag{5}$$

holds where ac_{wp} is a weakest c-preserving application condition for c and ac_{ws} is a weakest consistency-sustaining one.

Proof. Assume $m \models ac_{\text{ws}}$. Then, $G \Rightarrow_{m,p} H$ and $G \models c$ imply $H \models c$ since sustaining transformations are preserving [16, Theorem 1]. Hence, Eq. (2) in Lemma 2 implies $m \models ac_{\text{wp}}$.

In the other direction, assuming $m \models ac_{\text{wp}}$, we obtain $H \models c$ from $G \Rightarrow_{m,p} H$, $G \models c$, and Eq. (2). Then, the second part of the equivalence Eq. (4) in Lemma 2 is trivially true (there are no $\mathcal{M}$-morphisms $q' : C \hookrightarrow H$). In particular, this equivalence implies $m \models ac_{\text{ws}}$. □

Remark 2. Note that, in contrast to the definition of weakest preconditions in [13], our notion of *weakest* is based on matches, i.e., on morphisms at which the given rule is applicable, and not on arbitrary morphisms with domain L. This is, admittedly, not the most elegant definition from a theoretical point of view. Basing the definition on all morphisms with domain L, immediately guarantees that any two weakest application conditions (if there are any) are semantically equivalent. We only obtain equivalence on the restricted class of matches for the rule p; for other morphisms with domain L, two weakest application conditions can show different behavior. This means, they do not need to be semantically equivalent. In particular, according to our definition, weakest application conditions are not necessarily unique (up to semantic equivalence). However, our definition reflects our implementation and, arguably, the practical use of application conditions more appropriately.

This can be seen as follows: A weakest application condition with respect to *all morphisms* needs to evaluate to `true` for every morphism $m : L \to G$ that is not a match for the rule p.[2] Thus, analogous to the construction of ac_{pres} in Construction 3, to obtain a weakest application condition in this sense, one adds an antecedent to the "actual" application condition that ensures preservation or sustainment. In this case, the antecedent expresses the property of a morphism to be a match for the given rule. Computing such an antecedent is possible under quite general circumstances; see [13, Theorem 8]). Whereas the resulting application condition may have nicer theoretical properties, it will be more complex in general. Moreover, it has no practical effect for the applicability of rules compared to our definition: For a match, the added antecedent will evaluate to `true` and only the "actual" application condition determines if the rule is applicable or not. For a morphism that is not a match, the whole application condition will evaluate to `true` since the antecedent evaluates to `false` but the rule is not applicable anyhow. Besides, every tool applying transformation rules comes with an in-built facility to determine matches for rules, of course. Thus, additionally equipping rules with application conditions that check for applicability of a rule creates unnecessary overhead in practice. We therefore develop and implement our theory for matches instead of general morphisms. Adapting our results to the general case, however, is rather straightforward using the above mentioned translation of the applicability of a rule into an application condition.

4.2. Construction of weakest constraint-preserving or consistency-sustaining application conditions

In the following, we present the first simplifications that have already been explained informally at the example of typed graphs in the introduction of this section. The next definition expresses the notion that the set of elements created or deleted by a rule overlaps emptily with the set of constraint elements. This definition is based on the existence of morphisms, i.e., in a way that makes sense in any category. This definition allows us to present our simplifications in the context of an arbitrary $\mathcal{M}$-adhesive category where a reasonable notion of "element" is not necessarily available. This abstract definition conforms to the element-based intuition in the case of typed graphs with inheritance.

Definition 10 *(Trivial intersection).* In an $\mathcal{M}$-adhesive category $\mathcal{C}$, let c be a nested constraint and $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ a plain rule.

1. We say that c and p *intersect trivially w.r.t. deletion* if for every object C that occurs in the constraint c and every span $C \overset{s_C}{\hookleftarrow} S \overset{s_L}{\hookrightarrow} L$ where $s_L \in \mathcal{M}$ there exists a morphism $x : S \to K$ such that $l \circ x = s_L$ (compare Fig. 14).
2. Analogously, we say that c and p *intersect trivially w.r.t. creation* if for every object C that occurs in the constraint c and every span $C \overset{s_C}{\hookleftarrow} S \overset{s_R}{\hookrightarrow} R$ where $s_R \in \mathcal{M}$ there exists a morphism $x : S \to K$ such that $r \circ x = s_R$.
3. The constraint c and the rule p *intersect trivially* if they do so w.r.t. both, deletion and creation.

[2] This is seen analogously to the proof of Lemma 2 where we show that every weakest preserving application condition needs to evaluate to `true` for every match $m : L \to G$ where G is invalid.

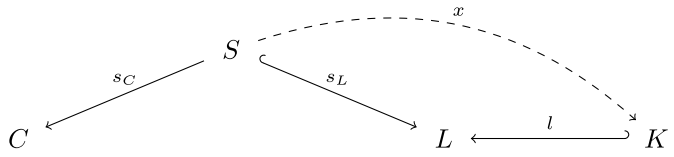


Fig. 14. Trivial intersection w.r.t. deletion between rule and constraint depicted for one object C.

In the context of typed graphs with inheritance, the above definition is equivalent to saying that none of the nodes of any graph occurring in the constraint has a type that lies in the type-clan of a node that is deleted or created by the rule, i.e., a node from $V_L \setminus V_K$ or $V_R \setminus V_K$, and similarly for the edges. This means that the above definition can be understood as saying that the constraint does not constrain elements that can be affected by applications of the rule. Considering our implementation, Henshin allows for type refinement in contrast to the theory, i.e., constraints are not evaluated with strong morphisms but just with injective ones. In such a context, one needs to adapt the above definition to just requiring that s_L is injective (and not $s_L \in \mathcal{M}$). This is equivalent to saying that the type-clans of nodes of the constraint graphs *intersect emptily* with the type-clans of all nodes that are deleted or created by the rule, i.e., nodes from $V_L \setminus V_K$ or $V_R \setminus V_K$ (and similar for the edges). The precise result is stated and proven as Lemma 3 in Appendix A. This result ensures that the following indeed generalizes our work from [21] and that our implementation, based on the comparison of the type-clans of elements, conforms to the theory.

We finally state the correctness of the simplifications that were informally explained in the introduction of this section in the following theorem.

Theorem 1 *(Constructing weakest application conditions). In an $\mathcal{M}$-adhesive category $\mathcal{C}$ with $\mathcal{E}'$-$\mathcal{M}$ pair factorization, let c be a nested constraint and $p = (L \hookleftarrow K \hookrightarrow R)$ be a plain rule.*

1. *If c and p intersect trivially, then $ac_{\text{wp}} = \texttt{true}$ is a weakest c-preserving application condition for p. Moreover, if c is a negative constraint, $\texttt{true}$ is also a weakest consistency-sustaining one.*
2. *If p is monotonic and c is a positive constraint, then $ac_{\text{wp}} = \texttt{true}$ is a weakest c-preserving application condition for p.*
   If p only deletes and c is a negative constraint, then $ac_{\text{wp}} = \texttt{true}$ is a weakest c-preserving and a weakest consistency-sustaining application condition for p w.r.t. c.
3. *If $c = \neg\exists\, C$, let*

$$rac := \text{Shift}(\emptyset \to R, c) = \neg(\bigvee_{i \in I} \exists\, P_i)$$

with morphisms $b'_i : C \hookrightarrow P_i \in \mathcal{M}$ and $a'_i : R \to P_i$. For each $i \in I$ let $C \xleftarrow{s^i_C} S_i \xhookrightarrow{s^i_R} R$ be the pullback of b'_i and a'_i. Let

$$rac_{\text{sp}} := \neg(\bigvee_{j \in J} \exists\, P_j)$$

where $J \subseteq I$ only includes those P_i where there is no morphism $x : S_i \hookrightarrow K$ such that $r \circ x = s^i_R$. Then

$$ac_{\text{sp}} := \text{L}(p, rac_{\text{sp}})$$

is a weakest consistency-sustaining application condition for p w.r.t. c.

The proofs of these three statements, given in Appendix A, follow a common pattern: In each of the cases, checking for the (non-)existence of objects occurring in the constraint is *sequentially independent* from applying the rule. Hence, the checking of the constraint for validity always gives the same result, no matter if done before or after rule application. This intuition is made precise in a technical lemma (Lemma 4) upon which our proof is based; this lemma is also presented in Appendix A.

We close this section by presenting two concrete examples for our simplifications. Both explain the examples from Sect. 2 in more details.

Example 6. Constraint no_region needs to be integrated into rule moveRegionFromStateToState since a region-edge is created by this rule and contained in this constraint. Fig. 5 shows the constraint-guaranteeing application condition. Examples 3 and 4 explain its computation. Our proposed optimizations lead to the result displayed in Fig. 6 by the application of Theorem 1, 3.: Except for the subcondition containing the uppermost graph, all other subconditions in Fig. 5 are omitted. The uppermost one has to be retained because the region-edge created by the rule overlaps with the region-edge of the constraint. The omitted subconditions do not only involve elements being local to the rule application but amount to traversing every existing FinalState which leads to global checks. To conclude, only one local check remains; six checks are

omitted, partly global checks. Theorem 1 guarantees that the result is a weakest consistency-sustaining application condition for moveRegionFromStateToState w.r.t. no_region.

Example 7. The constraint TransitionInRegion (Fig. 2) is not required to be integrated into the rule moveRegionFromStateToState. Theorem 1, 1. justifies this: The rule moveRegionFromStateToState does not have any effect on the validity of the constraint since its application neither deletes nor creates elements of types that occur in the constraint.

4.3. Formal comparison to Pres

In this section, we formally compare our optimization presented above to the construction Pres in [13] which was recalled in Construction 3. We first collect the formal results and close the section illustrating these using our running example. Even though it is not explicitly stated in [13], the construction Pres computes a weakest constraint-preserving application condition according to Definition 9 as the following fact shows.

Fact 4 *(ac_{pres} is a weakest constraint-preserving application condition). In an $\mathcal{M}$-adhesive category with $\mathcal{E}'$-$\mathcal{M}$ pair factorization, let a rule p and a constraint c be given. Then the application condition ac_{pres} as computed in Construction 3 is a weakest c-preserving application condition for p.*

Proof. Using Lemma 2 and inspecting the proof of Corollary 5 in [13] (where constraint preservation is proven), one sees that they show the even stronger equivalence stated in Eq. (2). This means, we only have to show that Eq. (3) also holds. But given any morphism $m : L \to G$ for p such that $G \not\models c$, by correctness of Shift (see Fact 1), $m \not\models \text{Shift}(\emptyset \to L, c)$. This implies $m \models ac_{\text{pres}} \coloneqq \text{Shift}(\emptyset \to L, c) \Rightarrow ac_{\text{gua}}$. □

Combining this with Corollary 1, we directly obtain the equivalence of constructed application conditions for matches (into valid objects).

Corollary 2 *(Equivalences between ac_{pres}, ac_{wp}, and ac_{ws}). In an $\mathcal{M}$-adhesive category with $\mathcal{E}'$-$\mathcal{M}$ pair factorization, let a plain rule p and a constraint c be given and ac_{pres} be the application condition computed according to Construction 3. Assuming that one of the simplifications 1. or 2. of Theorem 1 is applicable, i.e., $ac_{wp} = \texttt{true}$ is the resulting application condition, the equivalence*

$$m \models ac_{\text{pres}} \Leftrightarrow m \models ac_{\text{wp}}$$

holds for every match $m : L \to G$ for p. If c is negative and simplification 3. applies, the equivalence

$$m \models ac_{\text{pres}} \Leftrightarrow m \models ac_{\text{ws}}$$

holds for matches $m : L \to G$ for p where $G \models c$.

Proof. In the first case, since both application conditions are weakest c-preserving application conditions for p (according to Theorem 1 and Fact 4), we have $m \models ac_{\text{pres}} \Rightarrow m \models ac_{\text{wp}}$ and $m \models ac_{\text{wp}} \Rightarrow m \models ac_{\text{pres}}$ for every match m. The second claim follows analogously by Corollary 1, Theorem 1, and Fact 4. □

From a computational point of view, both approaches differ with respect to the way in which the resulting application condition is constructed. This has effects on the structural complexity of their output: Our approach omits parts of the application condition ac_{gua}, the constraint-guaranteeing application condition computed according to Construction 3, or even drops it altogether. It does so *during the computation of* ac_{gua} and, for graph-like structures, by *a simple comparison of the types of occurring graph elements*. In contrast, the construction of ac_{pres} involves an additional application of the Shift operator, and, instead of omitting parts of ac_{gua}, the number of occurring graphs is roughly doubled by the introduction of the implication. Further simplifying ac_{pres} to a semantically equivalent but structurally less complex application condition might often be possible but involves further costly computations. Moreover, to the best of our knowledge, no general algorithm for such a simplification has been suggested in the literature. However, it has to be conceded that our second and third simplifications have only been presented for special cases of constraints (even though important ones) whereas the construction of ac_{pres} is completely general.

From a semantical point of view, the above corollary gives us two cases to consider. In the case of the first two simplifications, our simplifications and the construction ac_{pres} of Habel and Pennemann are equivalent for all practical purposes (namely, equivalent on the class of matches for the rule). The above discussion shows that we obtain that result in a more efficient way (if simplifications are possible). For the third simplification, we obtain an application condition that shows the same behavior for matches to valid objects. Matches to invalid objects always satisfy ac_{pres}; see (the proof of) Fact 4. In contrast, by its defining property as sustaining application condition, such matches only satisfy ac_{ws} if the application does not introduce new violations. The desired kind of behavior will depend on the application scenario. However, there are a

Table 1
Overview of behavior of rules equipped with the different kinds of application conditions on the set of matches.

application condition	quality of input	applicability of equipped rule	quality of output
weakest guaranteeing (ac_{gua})	irrelevant	if and only if output is valid	valid
weakest preserving ($ac_{\text{pres}} = ac_{\text{wp}}$)	valid invalid	if and only if output is valid always	valid invalid or valid
weakest sustaining (ac_{ws})	valid invalid	if and only if output is valid if and only if amount of validity is sustained	valid no new violations

lot of scenarios, like model repair or editing, where the sustaining behavior will be of advantage as it allows for controlled work also on invalid models.

In Table 1, we summarize the disparate behavior of the different kinds of application conditions we discuss in this paper. For example, given a rule equipped with a weakest preserving application condition, the rule is applicable at a match (for its plain rule) to a valid model if and only if the output is valid. At a match to an invalid model, the rule is always applicable and might produce valid or invalid output.

We close this section by illustrating the above discussion with two concrete examples.

Example 8. Fig. 15 presents the application condition ac_{pres} of integrating the constraint `no_region` into the rule `moveRegionFromStateTOState` being computed according to Construction 3. The right column displays ac_{gua} which was already depicted in Fig. 5. The left column displays the additionally added antecedent that expresses that the model was already valid before rule application; it is computed using the Shift construction, i.e., by overlapping the LHS of the rule with the graph of the constraint in all possible ways. The resulting application condition ac_{pres} contains 14 graphs although we have integrated only one constraint into the rule. Of course, this application condition can be further simplified: The material implication has to be removed and then using De Morgan's law and distributivity yields a formula in conjunctive normal form. It consists of 7 clauses where each clause contains 8 literals. To simplify the particular clauses, subgraph isomorphism checks have to be performed. They allow to omit 6 of the 7 clauses as these turn out to be equivalent to `true`. In the end, reintroducing the material implication operator, we obtain an application condition that still has the left column of Fig. 15 as antecedent; however, the consequent only consists of the first graph of the right column. As soon as the input constraint is really nested, the simplification becomes even more difficult since particular clauses can no longer be simplified by simply checking for subgraph isomorphisms.

The antecedent ensures that the rule equipped with this application condition is applicable at every match in an invalid graph. Our constructed application condition ac_{ws} (Fig. 6) ensures that no new violation is introduced (or existing violation moved), i.e., no Region is (newly) moved into a FinalState. This property does not depend on the validity of the graph the rule is applied to.

Example 9. In Fig. 7, we show the TransitionInRegion-guaranteeing application condition for the rule moveRegionFromStateToState; let ac_{gua} denote this application condition in the context of the current example. In Example 7, we discuss how in that case we obtain `true` as weakest preserving application condition by our simplifications. Applying Construction 3 instead, results in $ac_{\text{gua}} \Rightarrow ac_{\text{gua}}$ which is clearly equivalent to `true` (as to be expected by Corollary 2). However, its computation is far more involved and the result has to be simplified, additionally. In particular, depending on the representation, it might be a complex task (involving several checks for graph isomorphism) to recognize the antecedent and consequent to actually be the same constraints (and hence the overall constraint to be equivalent to `true`).

4.4. Dealing with EMF's built-in negative constraints

In this section, we propose EMF-specific simplifications of application conditions. EMF has several built-in constraints [11]. Instance models that do not satisfy these EMF-constraints cannot even be opened in the EMF-model editor. Most of these constraints are negative, i.e., they forbid certain patterns in instances to exist. Concretely, cycles over containment edges, nodes with more than one container, and parallel edges, i.e., two edges of the same type between the same two nodes, are forbidden. Therefore, given an application condition ac of a rule p, each occurrence of a subcondition of the form $\exists C$ with C violating one of these EMF constraints, may be replaced by false without altering the meaning. We know that such patterns cannot appear in any EMF instance model. Thus, in the context of EMF, the result is semantically equivalent to the original application condition but may contain fewer subconditions.

Theorem 2 *(Correctness of EMF-specific simplifications). Let c be a graph condition over P and c' be the condition that results from replacing every occurrence of a subcondition $\exists(a : C_1 \hookrightarrow C_2)$ of c by `false` if the graph C_2 contains parallel edges or multiple incoming containment edges to the same node. Then an injective morphism $g : P \hookrightarrow G$ into an EMF-model graph G satisfies c if and only if it satisfies c'. In particular, if c is a graph constraint, any EMF-model graph G satisfies c if and only if it satisfies c'.*

Also the proof of this theorem is presented in Appendix A. It uses induction along the nesting structure of the constraint in the cases of parallel edges and multiple containment nodes. The same argument also applies in the case of finite containment cycles. But since containment cycles of arbitrary length cannot be expressed as graph constraints, the correctness of replacing their occurrence by `false` is intuitive but not amenable to a formal proof by induction.

Since the EMF specific simplifications do not change the semantics of the application condition in the context of EMF models, the following corollary is immediately clear.

Corollary 3 *(Semantic-preservation of EMF specific simplifications). Let ac be an application condition and ac' be the application condition that results from applying the simplifications from Theorem 2 to it. When the evaluation of ac and ac' is restricted to morphisms to EMF model graphs, if ac is a (weakest) constraint-preserving/guaranteeing/consistency-sustaining application condition for its rule with regard to a constraint c, so is ac'.*

We also illustrate these simplifications using an example that builds on our running example from Sect. 2.

Example 10. If one does not first apply Theorem 1 to the application condition in Fig. 5, Theorem 2 allows one to drop the third, sixth, and seventh subcondition from it and to replace each one with true (note that we replace negated formulas): In all three cases, the subcondition consists of a graph that contains a node with more than one container or parallel edges of the same type between the same two nodes. According to Example 6 and Corollary 3 the result is still a constraint-guaranteeing application condition as long as the rule is applied to an EMF model graph.

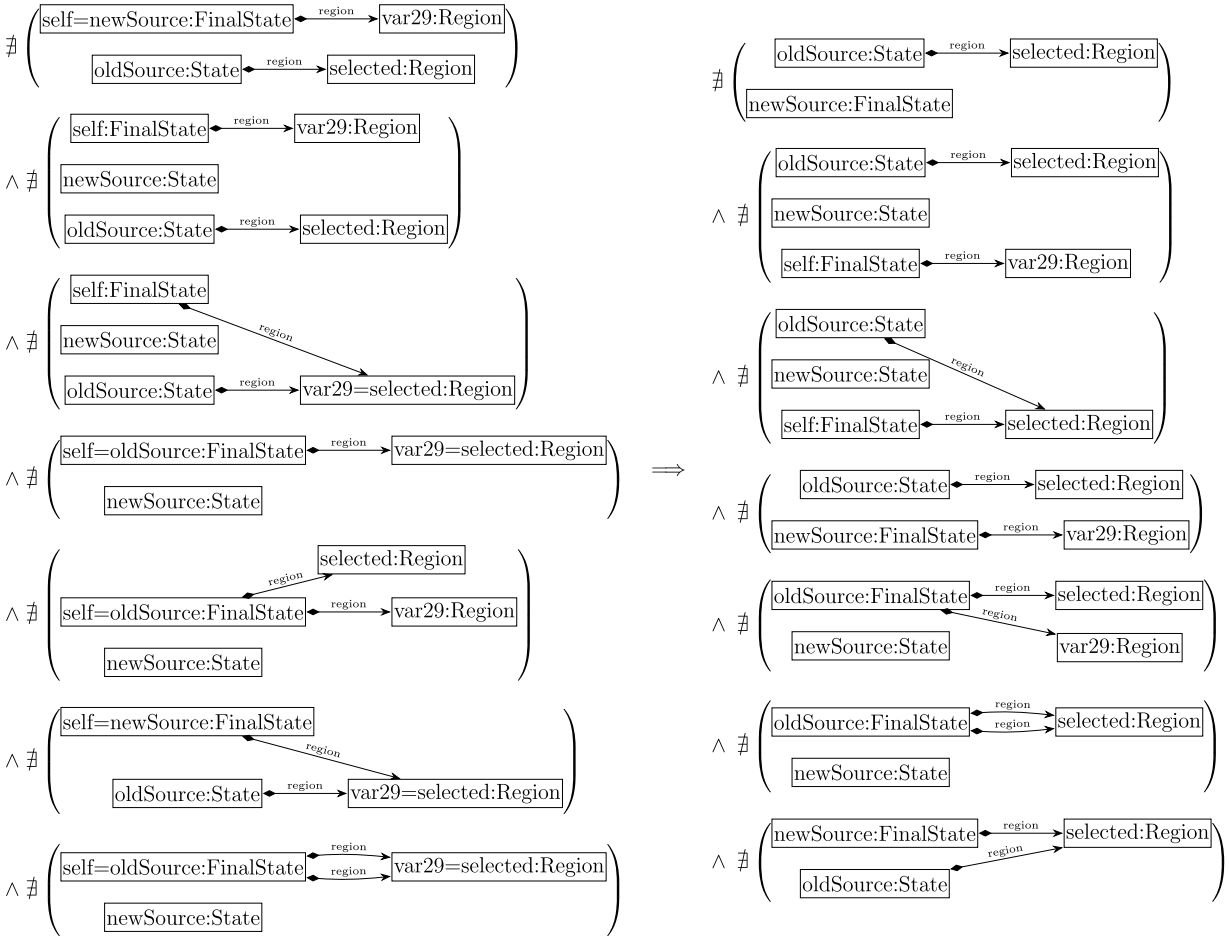


Fig. 15. The resulting c-preserving application condition ac_{pres} of the construction provided by Habel and Pennemann in [13] (Construction 3).

5. Tooling

We developed a tool, called *OCL2AC* [20], as an Eclipse plug-in implementing the existing theory [13,15] and our simplifications for constructing constraint-guaranteeing and -preserving application conditions from given transformation rules and constraints. *OCL2AC* consists of two main components:

- *OCL2GC* takes a set of OCL constraints as an input and automatically returns a set of semantically equivalent graph constraints as an output. This component performs several steps which are shown in Fig. 16 and are described as follows: In Step (1) an OCL constraint is prepared by refactorings. This is done to ease the translation process, especially to save translation rules for OCL constraints. Step (2) translates an OCL constraint to a so-called compact condition (a simple form of a condition) along the abstract syntax structure of OCL expressions. This translation largely follows the one in [15]. Step (4) completes the compact condition to a nested graph constraint being used to compute application conditions later on. The resulting nested graph constraint is simplified in Step (5) using equivalence rules.

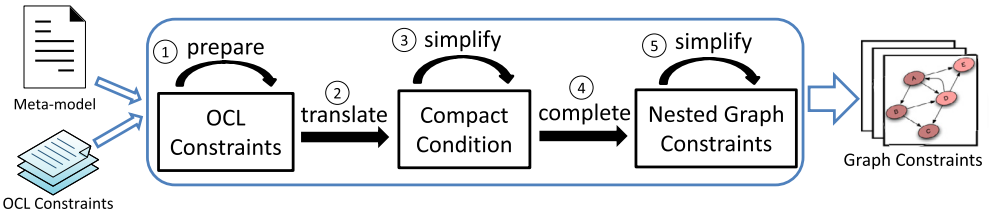


Fig. 16. *OCL2GC* component.

- *GC2AC* takes a transformation rule and a graph constraint as inputs and automatically returns the rule equipped with an application condition as output. The user can choose to compute the constraint-guaranteeing application condition ac_{gua} (according to Construction 3) or the weakest constraint-preserving (consistency-sustaining) application condition ac_{wp} (ac_{ws}) where the simplifications according to Theorem 1 are applied. If none of the preconditions of Theorem 1 holds, i.e., none of the simplifications can be applied, the user will be informed. The component performs several steps which are shown in Fig. 17 and are described as follows: In Step (1) the given graph constraint is prepared by refactorings. Step (2) performs one of the following options according to the user requirement: (a) immediately shifts a given graph constraint to the RHS of the given rule or (b) performs the optimization by applying Theorem 1 if possible. As an output, a new right application condition for the rule is computed. Step (3) translates a right application condition to the LHS of the given rule to get a new left application condition. It is translated by shifting it over the rule, i.e., by

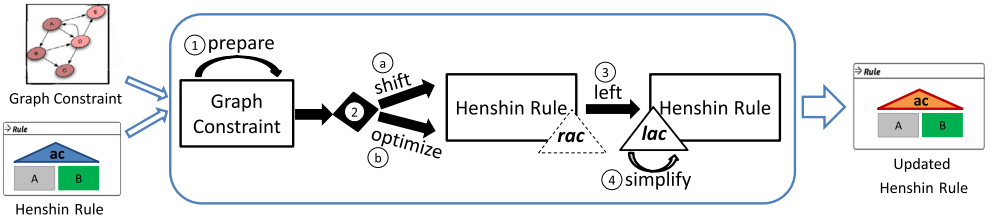


Fig. 17. *GC2AC* component.

applying the rule reversely to the right application condition. Step (4) simplifies the resulting left application condition by applying Theorem 2. The output is the original Henshin rule with the corresponding left application condition.

Each component can be used independently as an Eclipse-based tool. All the tooling developed can be downloaded from our website.[3]

Limitations *OCL2GC* is based on a formal approach which comes with the following limitations: The supported logic is two-valued and first-order and thus the expression oclIsUndefined and the operation iterate are not supported, for example. Moreover, there is no support to translate user-defined operations and there is only limited support to integrate constraints on attributes into Henshin rules that perform complex attribute computations. See [15] for more details on these limitations of the translation. In contrast, the theory for integrating graph constraints as application conditions into transformation rules is general in scope. This means, every graph constraint can be integrated into every rule and our implementation (i.e., the

[3] https://ocl2ac.github.io/home/.

component *GC2AC*) supports this. As discussed above and stated in Theorem 1, our simplifications can only be performed under certain preconditions. The implementation in *GC2AC* supports all proven simplifications.

6. Evaluation

In this section, we show the highlights of our evaluation; all details can be found in separate artifacts.[4]

Research questions (RQs) Our evaluation aims to answer the following RQs regarding the complexity and performance:

(RQ 1) *How complex are the resulting application conditions with and without optimizations? How does this compare to the complexity of the original graph constraints?*

To perform validity-preserving steps, there are two basic approaches: We either test for validity after each transformation step and rollback the step if its resulting model is not valid (*a posteriori* check) or the transformation is designed to perform validity-preserving steps only (*a priori* check). We, therefore, ask the following questions:

(RQ 2.1) *How fast is the* a priori *validity check compared to the* a posteriori *check?*
(RQ 2.2) *Does the optimization of application conditions improve the performance significantly?*

General set-up As an application case, we consider the scenario of in-place model transformations that should preserve a basic consistency such that the resulting instances can be opened in a domain-specific model editor throughout. In [5], Kehrer et al. derive consistency-preserving editing rules from a given meta-model. However, they support basic constraints like multiplicities only. More complex OCL constraints are left to future work. In their evaluation, this restriction has the most serious impact on the UML meta-model for Statecharts [30]. Out of 17 original constraints they identified 11 to be enforced in MagicDraw [22]. In total, they used 84 editing rules for Statecharts.

We translated those 11 OCL constraints into graph constraints and then integrated them as application conditions into the 84 rules.

7 valid test models of sizes between 800 to 16 000 elements (nodes and references) are used to conduct our performance experiments. These test models are synthetic containing copies of an initial valid model composing 5 objects of each non-abstract class of the meta-model. All evaluations were performed with a desktop PC, Intel Core i7, 16 GB RAM, Windows 7, Eclipse Neon, Henshin 1.4.

6.1. Evaluating complexity

In theory, the size of a computed application condition (the number of graphs) can grow exponentially in the worst case compared to the size of the original constraint [17]. In practice, however, the growth is moderate. Mainly due to node typing, many node overlappings are not possible. To find out how far this blow up of application conditions is a problem in practice, we conducted the following experiments considering the number of graphs as well as the number of nesting levels in application conditions. Additionally, we explore how far the complexity can be reduced using our optimization. Table 2 gives an overview of the results.

Integration without optimization Given the 11 OCL constraints of our application case, we translated them to graph constraints containing 2 to 10 graphs (36 in total) and integrated all of these in each of the 84 rules using OCL2AC (i.e., computing the guaranteeing application conditions). The newly added application conditions contain 77.3 graphs on average (with 36 being the best and 191 being the worst case) and 6 nesting levels. Thus, on average the number of graphs more or less doubles which is far better than could be suspected from theory. Nonetheless, the number of graphs is way too high and also the number of levels should be smaller in most cases. Hence, there is a clear need to further optimize the resulting application conditions.

Integration with optimization To find out how efficient our optimizations of application conditions are, we conducted the same experiment as above using our developed optimizer. In result, the average number of graphs in the application condition is 10.8 (with 0 being the best and 35 being the worst case), i.e., the complexity is reduced by factor 7 on average using our optimizer. Additionally, the deepest nesting level of 6 was often reduced to at most 2 levels. Theorem 1, 1 turns out to be the main reason behind this considerable loss of complexity: Instead of integrating 11 constraints into each rule, on average only 1.7 constraints are integrated into a rule.

Table 2 shows extreme cases: Considering all 84 rules and the 11 constraints, the best optimization was reached with rule create_Transition where the resulting application condition with 191 graphs was reduced to a condition with just one graph. One of the lowest optimizations came along with rule create_FinalState. Since it is overlapped with all the 11 constraints,

[4] These artifacts are available for download at the tool's website (https://ocl2ac.github.io/home/).

Table 2
Number of graphs of application conditions and deepest nesting levels before and after optimization (with emphasis on extreme cases).

Rule	without optimization		with optimization		
	#graphs	level	#graphs	level	#integrated constraints
create_Transition	191	6	1	1	1
create_FinalState	44	6	31	6	11
delete_Trigger	37	6	0	0	0
Average (84 rules)	77.3	6	10.8	2.6	1.7

the number of the resulting graphs is reduced by factor 1.4 only (using Theorem 1, 3). Rule delete_Trigger started with one of the lowest number of graphs in its application conditions. This condition is eliminated altogether using our optimization.

Across 10 runs, the average time of integrating the 11 graph constraints for statecharts into all 84 rules was 2.3 sec. without optimization and 1.03 sec. with optimization. In particular, calculation of our simplified application conditions is even faster than computing the guaranteeing ones. In both cases, calculating all needed application conditions for a given rule set is fast enough to be used in practice.

To answer RQ 1, given graph constraints with 2–10 graphs (3.2 on average) and 2–6 nesting levels (2.3 on average), non-optimized application conditions have 36–191 graphs (77.3 on average) and 6 nesting levels, while optimized ones have 0–35 graphs (10.8 on average) and 0–6 nesting levels (2.6 on average). Hence, condition sizes are considerably reduced (by factor 7 on average).

6.2. Evaluating performance

To answer RQ 2.1 and RQ 2.2, we set up two test scenarios comparing the runtime of *a posteriori* and *a priori* validity checks.

Experiment set-up Each test scenario (TS) consists of 15 test cases, one case for 15 selected rules (out of 84). These 15 rules are representative w.r.t. supported editing actions and rule size, in particular, they cover all kinds of editing actions. Their sizes range between 3 and 7 model elements. The average size of an application condition of the 15 rules is 56.4 graphs with nesting level 6 (without optimization) and 16.8 graphs with nesting level 3.1 (with optimization). A test case of TS 1 consists of first applying an original rule to a test model at a random match and then checking the validity of the resulting model (using (a) the EMF validator [31] configured to employ the OCLinEcore validator [32] to validate OCL constraints and (b) the OCL interpreter [33]). A test case of TS 2 consists of applying an updated rule (with (a) the guaranteeing and (b) the optimized application condition) to a test model at a random match. To eliminate effects stemming from the choice of match, each test case of a test scenario is performed 100 times. A test scenario in TS 1 (a) is performed in one run time session such that caching of information can be used advantageously. A second variant of TS 1 (a) performs each *a posteriori* check in a separate session making caching useless. All the test scenarios have been performed on all the 7 valid test models.

The average run times are measured over altogether 15 000 applications for each scenario. A timeout (TO) takes place if the average run time exceeds 5 minutes. To evaluate an OCL constraint using the OCL interpreter, the context object has to be given. Focusing on approach differences, the following times were excluded from the evaluation time: The time needed to find the context objects of all OCL constraints for the OCL interpreter, the loading time of a test model to any validator, and the time needed to roll back to the state of a test model after applying a rule whose resulting model does not satisfy the constraints.

Experiment results Table 3 shows the following results: *A posteriori* checking is performed in 3 variants. TS 1 (a) uses the EMF validator with and without caching mechanism since we noted the followings: In the first validation check, the EMF validator took 1.77 to 1.95 seconds to check a test model of size between 800 to 16 000, whereas in the next validation checks, it took only 5 to 63 milliseconds. Our understanding for this improvement is that the EMF validator saves the

Table 3
Average run time (in seconds) of a single rule application (and validation) over 15 test cases with 100 random matches each using models of varying size.

Scenario	(Caching)	Model size						
		800	**1 500**	**3 000**	**6 000**	**10 000**	**13 000**	**16 000**
TS 1(a)	(yes)	0.01	0.01	0.01	0.02	0.04	0.05	0.06
TS 1(a)	(no)	1.66	1.71	1.76	1.79	1.8	1.83	1.85
TS 1(b)	(no)	128.97	185.08	254.17	TO	TO	TO	TO
TS 2(a)	(no)	0.01	0.01	0.04	0.13	0.3	0.5	0.79
TS 2(b)	(no)	0.01	0.01	0.02	0.05	0.12	0.22	0.33

model state after the first validity check. Thus, in the next checks at the same run time session, the EMF validator is still able to reach the model in the cache such that only the elements affected by rule application are considered. Without caching, the average run times are less than 2 seconds; with caching they are even about two magnitudes faster. Using the OCL interpreter (TS 1 (b)) instead leads to run times over 2 minutes or even timeouts (after 5 min.). *A priori* checking is performed in two variants: In TS 2 (a) rules with non-optimized application conditions are used while the application conditions in TS 2 (b) are optimized. The run times of both variants are below 1 second and hence slightly better than in TS 1 (a) without caching. Using caching, however, TS 1 (a) is even faster. This consideration yields the answer to RQ 2.1. To answer RQ 2.2 we can see that using rules with optimized application conditions is two and a half times faster than without optimization. Almost all of the times our rules were applicable and thus the whole application condition of a rule was completely checked and evaluated. To conclude, we can state that scenarios TS 1 (a) and TS 2 are both fast enough to be usable in practice. However, a rollback step in the *a posteriori* approach (TS 1) may not always be feasible. For example, if the rollback step is defined by applying the inverse rule, this is might not always be applicable if the rule computes attribute values. Furthermore, in the *a posteriori* approach, the rule action is performed first which may cause dangerous situations in several fields such as a railway system, self-driving cars and an e-health system.

Threats to validity External validity can be questioned since we consider a limited number of OCL constraints and rules. For our performance experiments, we selected 15 out of 84 editing rules which are representative concerning their kinds (rules for creating, deleting, setting, unsetting, and moving model elements) and sizes. Moreover, we reduced the effect of the rules' matches by executing each rule at 100 matches chosen randomly from each given model. For performance evaluation, we restricted our studies to synthetic models. As we did not spot any performance bottleneck, we are convinced that using realistic models would not yield basically different results.

Concerning the considered OCL constraints it can be noticed that about half of them are simple negative constraints. However, all core features of OCL (logical operators, navigation expressions and collection operators) are covered and at least one rather complex constraint is included. And, more importantly, this kind of constraints seems to be quite typical for the chosen application case. Constraints required by model editors are often negative to forbid input that is not allowed anyway. Therefore, we are confident that the results are representative. Nevertheless, further case examples are interesting to be considered in the future. Furthermore, implementing the rule application and the validity checking using an incremental pattern matching engine (e.g., EMF IncQuery [34]) may even improve the performance of both approaches.

7. Related work

We first discuss work that is related to our suggested construction of application conditions, i.e., work that is related to our theoretical contribution (1). Works related to our application scenario and evaluation are discussed subsequently. In our evaluation, we compared the performances of *a posteriori* validity checks with that of *a priori* checks. Therefore, here we compare (2) to other approaches that ensure transformations to be constraint-preserving (*a priori* approach) and (3) to selected approaches that repair models (which may result from applying non-sustaining transformation rules; *a posteriori* approach).

7.1. Constructing and simplifying (application) conditions and constraints

Our construction of application conditions modifies the existing construction of constraint-guaranteeing ones. This construction was first suggested for negative constraints on graphs in [28] and has since been generalized to arbitrarily nested constraints in $\mathcal{M}$-adhesive categories by Habel and Pennemann [13]. Moreover, the last work also introduces preserving application conditions and notions of weakest constraints. We have compared our results to this standard construction in detail throughout this paper and, in particular, in Sect. 4.3.

The notion of *(direct) consistency sustainment* has only lately been introduced for a subclass of constraints (concretely, linear constraints in *alternating quantifier normal form*) in [16]. The authors offer static analysis techniques to recognize rules that are (directly) consistency sustaining but no construction. Our Theorem 1 offers such a construction for the case of negative constraints. It should be noted, however, that we already used consistency-sustaining rules in practical applications without having the formalization ready: Checking the rules employed for model repair and model generation in [26,35], it turns out that these rules are consistency-sustaining with respect to upper bound constraints.[5] Some of the classes of rules are even equipped with weakest consistency-sustaining application conditions: The negative application conditions (NACs) used for *additional-node-creation rules* in [26] are equivalent to the ones we derive with our construction when integrating upper bound constraints into the used rules. Similarly, edit rules used in [5] are consistency sustaining with respect to upper bound constraints. In particular, for the special case of atomic edit rules on graphs and upper bound constraints on edges, (automatic) constructions of (weakest) consistency-sustaining application conditions have already been used before this had been formalized. Our work allows to recognize them as such and offers an automatic construction under far more general circumstances ($\mathcal{M}$-adhesive categories compared to graphs, general rules compared to atomic ones, negative constraints

[5] Upper bounds of an EMF meta-model can easily be formalized as negative graph constraints.

compared to upper bound constraints). We conjecture that a systematic review of the practical use of application conditions (which is out of scope for this paper) would reveal that several employed application conditions are not only preserving or guaranteeing but indeed (weakest) consistency-sustaining ones.

Behr et al. [36] just proposed a construction of minimal constraint-preserving application conditions for negative constraints. It gives (almost) identical results to the simplifications we presented in [21] for graphs and generalize in our current work in Theorem 1.[6] Their work is also performed in the setting of $\mathcal{M}$-adhesive categories; however, they address the more general framework of sesqui-pushout rewriting [37] and develop their construction for a set of constraints directly. In contrast to our approach, they restrict to negative constraints and match morphisms from $\mathcal{M}$. Whereas the resulting application conditions are very similar (and each approach can be adapted to compute the application condition the other approach suggests), the ways in which the computations are performed, differ significantly. Behr et al. first decompose a negative constraint into spans that allow to reconstruct the forbidden pattern as pushout. One of the legs of this span is then embedded into the right morphism of the rule via a pullback; this is then followed by taking a pushout to compute the desired negative application condition. Some of the resulting NACs are further dropped as they can be seen to forbid already invalid input. Restricting to negative constraints, they also obtain the application condition `true` as result in the situations of our first and second simplification. In the situation of the third, they do not consider the case that the RHS of the rule may already violate the constraint (i.e., they won't compute `false` as part of the output) and, in case an object P_i occurring in the negative application condition already violates the original constraint, they additionally drop that part, too. In this way, they compute an application condition that is strictly between weakest preserving and weakest sustaining ones in the general case. Whereas they prove *minimality* of their application condition (in the sense that every part of it contributes to it being preserving), our notion of *weakness* characterizes our result logically. It is interesting future work to compare if one of the approaches can be implemented more efficiently.

Rules for semantic equivalences in graph constraints and conditions have been reported in several places [17,38,15] and their application can lead to considerable simplification in the structure of a constraint. There are also approaches and implementations simplifying OCL constraints, especially automatically generated ones [39,40]. Depending on the usage scenario, such simplifications could provide a valuable pre-processing step to our approach.

7.2. Ensuring transformation rules to be constraint-preserving

In [41,38], Azab, Pennemann, et al. introduced ENFORCe, a prototype implementation that can ensure the correctness of graph programs. Among others, it is also able to compute guaranteeing application conditions for rules (Construction 3). It supports (partially) labeled graphs, not EMF models. Moreover, there is no translation from OCL to graph constraints available. Clarisó et al. present in [42] how to calculate an application condition for a transformation rule and an OCL constraint, directly in OCL. The supported subset of OCL is slightly larger than in OCL2AC because, staying with OCL, they can support operations which are not first-order. The authors provide a correctness proof for the presented translation into application conditions. In addition, there is a partial implementation. To the best of our knowledge, our work is the only one which optimizes the resulting application conditions considerably.

Dyck et al. [43,44] present works that extend the one presented by Becker et al. [45]. Their goal is to verify whether a graph transformation system preserves graph consistency with regard to forbidden graph patterns. They can improve the performance of their check under very specific conditions, e.g., if the forbidden graph pattern is of a particular form and the negative application conditions of the rules of the system have no nesting and a common domain. One such improvement is done by reducing the number of the patterns that have to be checked. They check the ones which relate to the local match of the rule. This is similar to Theorem 1, 3 of our work where we remove irrelevant subconditions and keep the ones which relate to the rule match. However, the objective of their approach is not to automatically construct constraint-preserving rules but rather to verify the consistency of a set of transformation rules w.r.t. a set of forbidden graph patterns. If that fails to be the case, the rule system has to be adapted manually by the user. Additionally, EMF constraints are not considered.

Instead of adding an application condition to a transformation rule, Kosiol et al. [46] amend its action such that it becomes constraint-preserving or -guaranteeing. That approach works for a special class of constraints only and under suitable independence conditions of the involved rules and constraints. In the future, this could be combined with the approach presented here, thus preserving some kind of constraints by amending the actions of rules and others by computing suitable application conditions.

7.3. Model repair

When tolerating inconsistencies instead of preserving validity throughout, generally, an instance has to be *repaired* at some point, i.e., consistency has to be restored. This topic has drawn a lot attention in research in MDE; [47] offers a recent

[6] Because of the close timely concurrence of both papers, we only list the commonalities and differences between our approaches without formally proving them (as we do in the case of [13]). The following key technical observations back up our claims: The commonalities between our approaches are explained by the fact that trivial intersection with respect to deletion in our sense induces a pushout decomposition (in their sense) with left leg isomorphism. In both approaches, this leads to data which can be ignored when computing the negative application condition. Conversely, of all pushout decompositions that contribute to ignored data in their approach, we only ignore the ones with left leg isomorphism; this explains the differences.

literature review. The existing model repair techniques can be mainly categorized into syntactic and rule-based approaches on the one hand, and logic-based approaches, on the other hand. In the following we give an overview about some works of both approaches.

Syntactic and rule-based approaches In [48], the authors provide a syntactic approach for generating interactive repairs from first-order logic formulas that constrain UML documents. The user can choose from automatically generated repair actions when an inconsistency occurs. However, possible negative side effects are not taken into consideration. It is up to the user to find a way to repair the whole model (if any).

Nassar et al. [26,49] present a rule-based approach for repairing EMF models in an automated interactive way. First, the model elements are trimmed to satisfy the negative constraints, namely, the upper bounds and then the trimmed model is completed to satisfy the positive constraints, namely, the lower bounds without violating the upper bound constraints. The repair rules are automatically generated from multiplicity constraints imposed by a given meta-model. The repair rules with the algorithm are designed to sustain the model's validity w.r.t. the EMF constraints. The correctness of the algorithm and its termination are proven. A ready-to-use tool as an Eclipse-plug-in is developed, as well. However, supporting OCL constraints is left to future work.

Taentzer et al. [50,51] present an approach and a tool for change-preserving model repair based on graph transformation theory. From given edit operations taken from the edit history of two versions of consistent models and consistency-preserving operations, they calculate repair operations. These repair operations can be used to repair models when editing operations introduce violations. The performance of detecting repair operations is shown to be fast. In that approach consistency is not defined with regard to constraints but with regard to a grammar.

Habel and Sandmann have suggested several approaches to graph repair. While [52] focuses on deriving repair rules from constraints, [53] examines if a given set of rules can be used to repair constraints. Both approaches work for large classes of graph constraints. The works concentrate on formal properties like correctness, expressivity, and termination; neither approach is implemented.

Logic-based approaches Many approaches such as [54–57] provide support for automatic inconsistency resolution from a logical perspective. All these approaches provide automatic model completion; however, they may easily run into scalability problems (as stated in [47]). Since the input model is always translated into a logical model, the performance depends on the model size. Schneider et al. [58] provide two strategies to graph repair which rely on an existing graph generation algorithm for graph constraints implemented in AutoGraph. The input graph is represented as a constraint and then it is given to AutoGraph with other constraints to generate a consistent graph. As a result they provide a complete overview over possible least change repairs. However, if the input graph and these constraints are mutually exclusive, no consistent graph can be found. The implementation is not based on EMF. AutoGraph cannot support all EMF constraints since some of them are not first order.

8. Conclusion

Application scenarios where each graph transformation step has to preserve the validity of models w.r.t. given constraints are needed in practice. As the construction of application conditions by Habel and Pennemann in [13] yields constraint-guaranteeing ones and assuming that the preservation of graph validity is already sufficient, the resulting application conditions can be considerably optimized. We developed several techniques (in Theorem 1 and Theorem 2) to construct optimized constraint-preserving application conditions and implemented them on top of OCL2AC. We proved that the first group of simplifications lead to weakest constraint-preserving or even weakest consistency-sustaining application conditions and the second group preserves the semantics of the simplified application condition in the context of EMF. In our evaluation, the usability of OCL2AC was investigated, with and without optimization. The evaluation results show that OCL2AC can lead to quite large application conditions which can be significantly optimized by factor 7 (on average) using our developed techniques. Accordingly, while the performance results of correct graph transformations are good in general, applying rules with optimized application conditions is shown to be ca. 2.5 times faster than applying non-optimized ones.

In future, we intend to further optimize resulting application conditions by identifying redundant subconditions, by checking negative invariants of modeling languages, and by considering a larger class of constraints. One of our goals is to obtain understandable application conditions. This work is already an essential step into that direction. Moreover, our optimization of conditions could have some interesting applications beyond MDE. We are interested, e.g., in assessing if our ideas can be beneficially integrated into proof systems.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

We are grateful to Annegret Habel, Christian Sandmann, and Steffen Vaupel for their helpful comments on the conference version of this paper.

Appendix A. Proofs and technical lemmas

We first state and prove a lemma that shows the equivalence of the morphism-based definition of trivial intersection (Definition 10) with the element-based intuition behind it.

Lemma 3 *(Characterization of trivial intersection). Let $\mathcal{C}$ be the category of typed graphs with inheritance. Then a plain rule $p = (L \hookleftarrow K \hookrightarrow R)$ and a nested graph constraint c intersect trivially if and only if 1.) no node of any graph C that occurs in c lies in the type clan of any node of $V_L \setminus V_K$ or $V_R \setminus V_K$ (short for the set-theoretic difference $V_L \setminus l_V(V_K)$) and 2.) no edge of any graph C that occurs in c shares its type with any of those edges of $E_L \setminus E_K$ or $E_R \setminus E_K$ whose source and target node stems from V_K. This means, the equations*

$$\forall n \in (V_L \setminus V_K) \left(clan_I(type_{LV}(n)) \bigcap \left(\bigcup_{C \in c} \bigcup_{n' \in V_C} type_{CV}(n') \right) = \emptyset \right) \quad , \tag{A.1}$$

and

$$\forall e \in (E_L \setminus E_K) \left((src_L(e) \in l_V(V_K) \wedge tar_L(e) \in l_V(V_K)) \Rightarrow \left(type_{LE}(e) \bigcap \left(\bigcup_{C \in c} \bigcup_{e' \in E_C} type_{CE}(e') \right) = \emptyset \right) \right) \tag{A.2}$$

and the according equations for the RHS hold, where $C \in c$ abbreviates the occurring of C in c.

In contrast, if we adapt the definition of a trivial intersection (Definition 10) for typed graphs with inheritance by not requiring $s_L \in \mathcal{M}$ but requiring spans $C \xleftarrow{s_C} S \xhookrightarrow{s_L} L$ where s_C, s_L are ordinary graph morphisms, i.e., not typed ones, s_L is injective, and for each $y \in S$ (node or edge) there exists an element t in the type graph such that $t \leq type_L(s_L(y))$ and $t \leq type_C(s_C(y))$, then a plain rule $p = (L \hookleftarrow K \hookrightarrow R)$ and a nested graph constraint c intersect trivially according to the adapted definition if and only if 1.) for all nodes of $V_L \setminus V_K$ and of $V_R \setminus V_K$ their type clans intersect emptily with the type clan of any node of any graph C that occurs in c and 2.) no edge of any graph C that occurs in c shares its type with any of those edges of $E_L \setminus E_K$ or $E_R \setminus E_K$ whose source and target node stems from V_K. This means, the equation

$$\forall n \in (V_L \setminus V_K). \left(clan_I(type_{LV}(n)) \bigcap \left(\bigcup_{C \in c} \bigcup_{n' \in V_C} clan_I(type_{CV}(n')) \right) = \emptyset \right) \tag{A.3}$$

and Eq. (A.2) hold (again, also the according equations for the RHS).

Proof. We only show the second statement as both proofs are very similar. First, assume that Eqs. (A.2) and (A.3) hold. Let $C \xleftarrow{s_C} S \xhookrightarrow{s_L} L$ be a span of ordinary graph morphisms such that s_L is injective, C is some typed graph occurring in c, and for each $y \in S$ (node or edge) there exists an element t in the type graph such that $t \leq type_L(s_L(y))$ and $t \leq type_C(s_C(y))$. If S is empty, the empty morphism from S to K has the desired property. Therefore, let $n \in V_S$ be a node. By assumption on the morphisms s_C and s_L there exists an element t in the type graph such that $t \leq type_L(s_{L,V}(n))$ and $t \leq type_C(s_{C,V}(n))$ which means that t lies in the intersection of the type clans of $s_{L,V}(n)$ and $s_{C,V}(n)$. By Eq. (A.3) this means that $s_{L,V}(n)$ has a (unique) preimage in V_K under l_V; this can be used to define the morphism x by setting $x(n) \coloneqq l_V^{-1}(s_{L,V}(n))$. If $e \in E_S$ is an edge, the computation for nodes shows that both the source and target node of $s_{L,E}(e)$ have a preimage under l_V. Then by Eq. (A.2) and the assumption that for each $y \in S$ (node or edge) there exists an element t in the type graph such that $t \leq type_L(s_L(y))$ and $t \leq type_C(s_C(y))$ also $s_{L,E}(e)$ has a preimage under $l_E(E_K)$. Again, this can be used to define $x(e) \coloneqq l_E^{-1}(s_{L,E}(e))$. By construction, x is a graph morphism and $l \circ x = s_L$ when l is just considered as graph morphism.

In the other direction assume that Eq. (A.3) or Eq. (A.2) is violated. We construct a span $C \xleftarrow{s_C} S \xhookrightarrow{s_L} L$ such that there is no graph morphism $x : S \to K$ with $l \circ x = s_L$. If there is a graph C that occurs in the constraint c such that the type clan of one of its nodes n' intersects with the one of a node n from $V_L \setminus V_K$, let S be the graph that just consists of one node. This node is mapped under $s_{C,V}$ to n' and under $s_{L,V}$ to n. By construction, s_C, s_L are graph morphisms, s_L is injective, and for each $y \in S$ (node or edge) there exists an element t in the type graph such that $t \leq type_L(s_L(y))$ and $t \leq type_C(s_C(y))$ (since the type clans of n and n' have a non-empty intersection). But there cannot be a morphism $x : S \to K$ such that $l \circ x = s_L$ since $n \notin l_V(V_K)$. Similarly, if Eq. (A.2) is violated, we take S to be a graph consisting of an edge together with its source and target node (a single node if the according edge e from $E_L \setminus E_K$ has the same source and target node) and use exactly the same idea to define morphisms $s_C : S \to C$ and $s_L : S \to L$ such that there is no morphism $x : S \to K$ with $l \circ x = x_L$. □

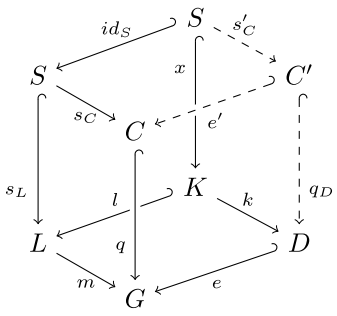
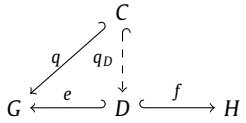


Fig. A.18. Constructing $q' : C \hookrightarrow D$.

The next technical lemma is the key ingredient to prove that our simplifications result in weakest constraint-preserving application conditions.

Lemma 4 *(Trivial intersection for transformation steps). In an $\mathcal{M}$-adhesive category $\mathcal{C}$, let a nested constraint c and a plain rule $p = (L \hookleftarrow K \hookrightarrow R)$ be given. Let $G \overset{e}{\hookleftarrow} D \overset{f}{\hookrightarrow} H$ be a span that arises by an application of the rule p to G at some match m.*

1. *If c and p intersect trivially w.r.t. deletion, for every $\mathcal{M}$-morphism $q : C \hookrightarrow G$ (where C is some object occurring in the constraint c), there is an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ such that $e \circ q_D = q$ (compare the depiction to the right).*
2. *Analogously, if c and p intersect trivially w.r.t. creation, for every $\mathcal{M}$-morphism $q' : C \hookrightarrow H$ (where C is some object occurring in the constraint c), there is an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ such that $f \circ q_D = q'$.*



In particular, if c and p intersect trivially and C is an object occurring in c, there are one-to-one correspondences between the $\mathcal{M}$-morphisms from C to G and from C to D and from C to H.

Proof. Let c and p intersect trivially with respect to deletion, $G \overset{e}{\hookleftarrow} D \overset{f}{\hookrightarrow} H$ be a span that arises by application of rule p to G at some match m, C be some object occurring in c, and $q : C \hookrightarrow G \in \mathcal{M}$. Compare Fig. A.18 for the following reasoning. The pullback of m and q exists and results in a span $L \overset{s_L}{\hookleftarrow} S \overset{s_C}{\rightarrow} C$ with $s_L \in \mathcal{M}$ since $\mathcal{M}$ is closed under pullbacks. Since c and p intersect trivially w.r.t. deletion, there exists a morphism $x : S \rightarrow K$ such that $l \circ x = s_L$; in particular, $x \in \mathcal{M}$ by decomposition of $\mathcal{M}$-morphisms. Since l is a monomorphism and $l \circ x = s_L$, pulling back s_L and l results in the span $S \overset{id_S}{\hookleftarrow} S \overset{x}{\hookrightarrow} K$. Similarly, pulling back q and e, and k and q_D subsequently, completes the cube depicted in Fig. A.18. The cube has a pushout as bottom square, pullbacks as side squares, and $s_L, q, q_D \in \mathcal{M}$ and hence, by the vertical weak van Kampen property, the top square is a pushout. Since pushout complements for a composition of morphisms where the first one is from $\mathcal{M}$ are unique in $\mathcal{M}$-adhesive categories, $C' = C$ and $e' = id_C$ (up to isomorphism). Hence, there exists an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ such that $e \circ q_D = q$.

The proof in case of trivial intersection w.r.t. creation is completely symmetric. In particular, if c and p intersect trivially and C is some object occurring in C, for every $\mathcal{M}$-morphism $q : C \hookrightarrow G$ there is an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ such that $e \circ q_D = q$ and for every $\mathcal{M}$-morphism $q' : C \hookrightarrow H$ there is an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ such that $f \circ q_D = q'$.

In case one assumes the definition of trivial intersection that was adapted to the behavior of Henshin, i.e., one allows for type-refinement in the morphisms q (and hence also s_L) basically the same proof works. Given a match $m : L \rightarrow G$ and an injective morphism $q : C \hookrightarrow G$ (that is not necessarily strong), we can "forget" the typing and perform the same computation as above for plain graphs. The pullback of m and q results in a span of graph morphisms $L \overset{s_L}{\hookleftarrow} S \overset{s_C}{\rightarrow}$ where s_L is injective and, since q and m are typed graph morphisms, for each $y \in S$ (node or edge) there exists an element t in the type graph such that $t \leq type_L(s_L(y))$ and $t \leq type_C(s_C(y))$. Since the category of graphs is even adhesive [24], we can perform the exact same computation as above (we do not need the assumption $q, s_L, q_D \in \mathcal{M}$ in an adhesive category) and receive an injective graph morphism $q_D : C \rightarrow D$. Since $e : D \hookrightarrow G$ is type-strict, q_D even is a typed graph morphism. □

We now prove our first theorem.

Proof of Theorem 1. Since every match satisfies `true`, the left sides of the equivalences Eqs. (2) and (4) in Lemma 2 are always true and, hence, the implications from right to left always are. Thus, for the statements 1. and 2. of the Theorem, it

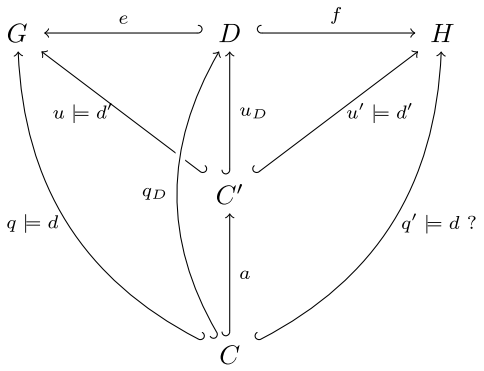


Fig. A.19. Satisfaction of condition d by q'.

suffices to show that `true` is a constraint-preserving, resp. directly consistency-sustaining application condition. Throughout the proof, let $G \xleftarrow{e} D \xhookrightarrow{f} H$ be the span induced by the transformation $G \Rightarrow_{m,p} H$.

1. Let $G \Rightarrow_{m,p} H$ be a transformation via rule p at match m. We first show by structural induction that for any condition d over any object C occurring in c, an $\mathcal{M}$-morphism $q : C \hookrightarrow G$ satisfies d if and only if $q' : C \hookrightarrow H$ satisfies d where $q_D : C \hookrightarrow D$ is the $\mathcal{M}$-morphism with $q' = f \circ q_D$ and $q = e \circ q_D$ that exists according to Lemma 4.
   First, if $d = \texttt{true}$, every $\mathcal{M}$-morphism $q : C \hookrightarrow G$ satisfies d as well as every $\mathcal{M}$-morphism $q' : C \hookrightarrow H$. Let $d = \exists\,(a : C \hookrightarrow C', d')$ and $q : C \hookrightarrow G \in \mathcal{M} \models d$. By definition, there exists an $\mathcal{M}$-morphism $u : C' \hookrightarrow G$ such that $u \circ a = q$ and $u \models d'$. By induction hypothesis, $u' \models d'$ where $u' := f \circ u_D : C' \hookrightarrow H$ and u_D is the $\mathcal{M}$-morphism with $e \circ u_D = u$ that is guaranteed to exist by Lemma 4. Then, by definitions of q_D and u_D and since $q \models d$ (compare Fig. A.19 for the calculations),

$$\begin{aligned} e \circ q_D &= q \\ &= u \circ a \\ &= e \circ u_D \circ a \ . \end{aligned}$$

   Since e is a monomorphism, this implies $q_D = u_D \circ a$. Using this in the next calculation, one receives

$$\begin{aligned} q' &= f \circ q_D \\ &= f \circ u_D \circ a \\ &= u' \circ a \ . \end{aligned}$$

   Thus, $q' \models d$. The opposite direction is completely analogous (since f is a monomorphism as well) and the induction steps for the Boolean operators are standard.
   Let $g : \emptyset \to G$, $g' : \emptyset \to H$, and $G \models c$. Now, if the original constraint $c = \texttt{true}$, $g \models c \Leftrightarrow g' \models c$. If c is nested, i.e., $c = \exists(\emptyset \to C, d)$, $g \models c \Leftrightarrow g' \models c$ by the considerations above. By structural induction (on Boolean operators) the same is true for any Boolean combinations of constraints and hence for any constraint. Now, $g \models c \Rightarrow g' \models c$ means that `true` is c-preserving.
   Moreover, in case $c = \neg\exists\, C$, for every $\mathcal{M}$-morphism $q' : C \hookrightarrow H$, we obtain an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ with $f \circ q_D = q'$ (like repeatedly used above). In particular, `true` is directly consistency sustaining.
2. The second case is proven basically in the same way by observing that a monotonic rule and a positive constraint intersect trivially w.r.t. deletion while a rule which only deletes and a negative constraint intersect trivially w.r.t. creation.
3. Let $c = \neg\exists\, C$, $G \Rightarrow_{m,p} H$, and first assume $m \models ac_{\text{ws}}$. To show the first direction of the equivalence in Eq. (4) in Lemma 2, we have to show that for any $q' : C \hookrightarrow H \in \mathcal{M}$ there is a morphism $q_D : C \to D$ such that $f \circ q_D = q'$. Hence, assume such an $\mathcal{M}$-morphism $q' : C \hookrightarrow H$ to exist. By existence of an $\mathcal{E}'$-$\mathcal{M}$ pair factorization, there are an object P_i and morphisms $(a'_i : R \to P_i, b'_i : C \to P_i) \in \mathcal{E}'$ and $g' : P_i \hookrightarrow H \in \mathcal{M}$ such that $n = g' \circ a'_i$ and $q' = g' \circ b'_i$ (see Fig. A.20). Moreover, $b'_i \in \mathcal{M}$ by decomposition of $\mathcal{M}$-morphisms. In particular $(a'_i, b'_i) \in \mathcal{F}$ which means that $\neg\exists(a'_i : R \to P_i)$ is one of the formulas constituting rac. First, g' witnesses that $n \models \exists(a'_i : R \to P_i)$ and therefore Fact 2 implies that $i \notin J$, i.e., $\neg\exists(a'_i : R \to P_i)$ is none of the formulas constituting rac_{ws} (otherwise $m \not\models ac_{\text{ws}}$). Secondly, computing the pullback of the morphisms a'_i and b'_i results in a span $R \xleftarrow{s^i_R} S_i \xrightarrow{s^i_C} C$ with $s^i_R \in \mathcal{M}$. By $i \in I \setminus J$ and the construction of J, this implies that there is a morphism $x : S_i \hookrightarrow K$ such that $r \circ x = s^i_R$. In particular, as in the proof of Lemma 4, there is an $\mathcal{M}$-morphism $q_D : C \hookrightarrow D$ with $f \circ q_D = q'$.

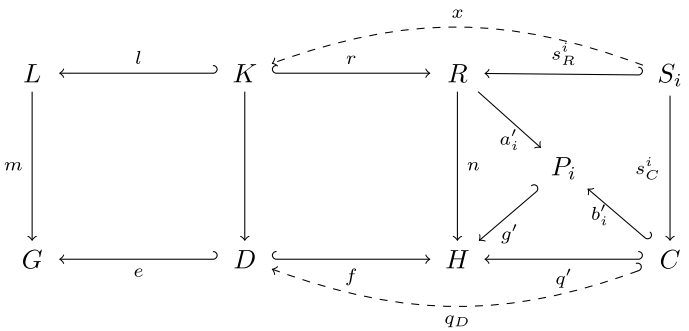


Fig. A.20. Existence of satisfying morphism.

For the second direction, assume that $\forall q' : C \hookrightarrow H . \exists q_D : C \hookrightarrow D . f \circ q_D = q'$. Additionally, for contradiction, assume that $G \Rightarrow_{p,m} H$ but $m \not\models ac_{\text{ws}}$. Then, by Fact 2 and the construction of ac_{ws}, $n \not\models rac_{\text{ws}}$. Hence, there is some $i \in J$ such that there exists an $\mathcal{M}$-morphism $g' : P_i \hookrightarrow H$ with $p \circ a_i' = n$. By definition of J then, there is no morphism $x : S_i \to K$ with $r \circ x = s_R'$ where, again, S_i is computed as pullback. But for $q' := g' \circ b_i' : C \hookrightarrow H$ our assumption implies the existence of a morphism $q_D : C \hookrightarrow D$ such that $f \circ q_D = q'$. Moreover, the pushouts defining the transformation are also pullbacks (as pushouts along $\mathcal{M}$-morphisms). Then, because of $f \circ q_D \circ s_C^i = q' \circ s_C^i = n \circ s_R^i$, the universal property of K as pullback object imply the existence of a morphism $x : S_i \to K$ such that $r \circ x = s_R^i$ which contradicts our assumption.
Summarizing, by Lemma 2 we conclude that ac_{ws} is a weakest directly consistency-sustaining application condition. □

Remark 3. As Lemma 4 also holds for the type-refining evaluation of application conditions, the proofs for the first two simplifications carry over immediately. For the third case, again a little adaption is necessary. Instead of working with the $\mathcal{E}'$-$\mathcal{M}$ pair factorization that splits a pair of morphisms with the same codomain into a pair of jointly surjective morphisms followed by an injective and type-strict one, we use another (non-unique) factorization, namely into a pair of jointly surjective morphisms followed by an injective one (not necessarily type-strict). Then, exactly the same proof works. The proof of the correctness of the Shift-operator does not depend on the uniqueness of the used factorization; a unique factorization just guarantees that the constructed class $\mathcal{F}$ is minimal to serve its purpose. The non-uniqueness of that factorization comes about in the following way: If there are two morphisms $f_1 : A_1 \to C$, $f_2 : A_2 \to C$ and there is a node $n \in V_C$ with preimages under f_1 and f_2 that have higher types in A_1 and in A_2, since m is not required to be type-strict, there can exist several choices for a type for this node n in K. But given the morphisms f_1, f_2, the possible choices for K form a partial order (with $K \geq K'$ when there exists a morphism $K \to K'$ making the obvious diagram commute). Choosing the maximal elements of that partial order results in a minimal class $\mathcal{F}$ again. In our implementation this is reflected by the fact that when computing the shift of a condition along a morphism (Construction 1), our class $\mathcal{F}$ consists of pairs of morphisms which are jointly surjective, both injective (adaption to injective matching of rules in Henshin), and where both are allowed to refine types (adaption to the evaluation of conditions in Henshin). Moreover, whenever there are pairs $b' : C \to C', a' : P' \to C'$ and $b'' : C \to C'', a' : P' \to C''$ (compare Fig. 11) such that there exists a morphism $c : C' \to C''$ with $c \circ a' = a''$ and $c \circ b' = b''$, we only consider the pair (b', a') in $\mathcal{F}$ which again results in a minimal class $\mathcal{F}$.

The last proof is the one of our second theorem.

Proof of Theorem 2. We first show that this replacement results in a graph condition again: Since all morphisms are injective, if a subcondition $\exists(a : C_1 \hookrightarrow C_2)$ is replaced by `false` because of such a violation, so are all subconditions dependent of C_2 in the tree structure of the condition: These subconditions contain the same violation.

We prove the general statement using structural induction.

The statement holds for $c =$`true` since `true' = true` and $g \models$`true` for every injective morphism g.

Let $c = \exists(a : P \hookrightarrow C, d)$ be a condition and, by induction hypothesis, $q \models d \Leftrightarrow q \models d'$ for each injective morphism q from C to any EMF-model graph. First, if C neither contains parallel edges nor multiple incoming containment edges to the same node, then $c' = \exists(a : P \hookrightarrow C, d')$. Now, for every injective morphism $g : P \to G$, where G is any EMF-model graph

$$\begin{aligned} g \models c &\Leftrightarrow \exists\, q : C \hookrightarrow G, \text{ s.t. } q \circ a = g \text{ and } q \models d \\ &\Leftrightarrow \exists\, q : C \hookrightarrow G, \text{ s.t. } q \circ a = g \text{ and } q \models d' \\ &\Leftrightarrow g \models c' \ . \end{aligned}$$

Secondly, if C contains parallel edges or multiple incoming containment edges to the same node, then $c' =$ `false`. Therefore, no injective morphism $g : P \hookrightarrow G$ satisfies c', where G is any graph. But since no EMF-model graph G contains parallel edges or multiple incoming containment edges to the same node, there does not exist any injective morphism $q : C \to G$

into any EMF-model graph G. Hence, there never exists an injective morphism q s.t. $q \circ a = g$ and $q \models d'$. In summary, $g \not\models c \Leftrightarrow g \not\models c'$ for all injective morphisms $g : P \hookrightarrow G$ for any EMF-model graph G.

The induction steps for Boolean operators are routine. □

References

[1] S. Sendall, W. Kozaczynski, Model transformation: the heart and soul of model-driven software development, IEEE Softw. 20 (5) (2003) 42–45, https://doi.org/10.1109/MS.2003.1231150.

[2] R. Balzer, Tolerating inconsistency, in: Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society Press, 1991, pp. 158–165, http://portal.acm.org/citation.cfm?id=256664.256748.

[3] A. Egyed, Instant consistency checking for the UML, in: Proceedings of the 28th International Conference on Software Engineering, New York, 2006, pp. 381–390.

[4] B. Becker, L. Lambers, J. Dyck, S. Birth, H. Giese, Iterative development of consistency-preserving rule-based refactorings, in: J. Cabot, E. Visser (Eds.), Theory and Practice of Model Transformations, Springer, Berlin, 2011, pp. 123–137.

[5] T. Kehrer, G. Taentzer, M. Rindt, U. Kelter, Automatically deriving the specification of model editing operations from meta-models, in: P.V. Gorp, G. Engels (Eds.), Theory and Practice of Model Transformations, ICMT 2016, Springer, Heidelberg, 2016, pp. 173–188.

[6] F. Steimann, M. Frenkel, M. Voelter, Robust projectional editing, in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, ACM, New York, 2017, pp. 79–90.

[7] C. Krause, H. Giese, Probabilistic graph transformation systems, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Graph Transformations, Springer, Berlin and Heidelberg, 2012, pp. 311–325.

[8] H. Giese, S. Glesner, J. Leitner, W. Schäfer, R. Wagner, Towards verified model transformations, in: Proc. of the 3rd International Workshop on Model Development, Validation and Verification, MoDeV2a, Genova, 2006, pp. 78–93.

[9] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Springer, Berlin and Heidelberg and New York, 2006.

[10] H. Ehrig, C. Ermel, U. Golas, F. Hermann, Graph and Model Transformation – General Framework and Applications, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2015.

[11] E. Biermann, C. Ermel, G. Taentzer, Formal foundation of consistent EMF model transformations by algebraic graph transformation, Softw. Syst. Model. 11 (2) (2012) 227–250, https://doi.org/10.1007/s10270-011-0199-7.

[12] A. Rensink, Representing first-order logic using graphs, in: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Eds.), Graph Transformations, Springer, Berlin, 2004, pp. 319–335.

[13] A. Habel, K.-H. Pennemann, Correctness of high-level transformation systems relative to nested conditions, Math. Struct. Comput. Sci. 19 (2009) 245–296, https://doi.org/10.1017/S0960129508007202.

[14] OMG, Object constraint language, http://www.omg.org/spec/OCL/, 2014.

[15] H. Radke, T. Arendt, J.S. Becker, A. Habel, G. Taentzer, Translating essential OCL invariants to nested graph constraints for generating instances of meta-models, Sci. Comput. Program. 152 (2018) 38–62, https://doi.org/10.1016/j.scico.2017.08.006.

[16] J. Kosiol, D. Strüber, G. Taentzer, S. Zschaler, Graph consistency as a graduated property. Consistency-sustaining and -improving graph transformations, in: F. Gadducci, T. Kehrer (Eds.), Graph Transformation, ICGT 2020, Springer International Publishing, Cham, 2020.

[17] K.-H. Pennemann, Generalized Constraints and Application Conditions for Graph Transformation Systems, Diplomarbeit, Department für Informatik, Universität Oldenburg, 2004, https://bit.ly/2T4RV0A.

[18] H. Ehrig, U. Golas, F. Hermann, Categorical frameworks for graph transformation and HLR systems based on the DPO approach, Bull. EATCS 102 (2010) 111–121, http://eatcs.org/beatcs/index.php/beatcs/article/view/158.

[19] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: advanced concepts and tools for in-place EMF model transformations, in: D.C. Petriu, N. Rouquette, O. Haugen (Eds.), Proc. of MODELS 2010, Springer, Berlin and Heidelberg, 2010, pp. 121–135.

[20] N. Nassar, J. Kosiol, T. Arendt, G. Taentzer, OCL2AC. Automatic translation of OCL constraints to graph constraints and application conditions for transformation rules, in: L. Lambers, J. Weber (Eds.), Proc. of ICGT 2018, Springer, Cham, 2018, pp. 171–177.

[21] N. Nassar, J. Kosiol, T. Arendt, G. Taentzer, Constructing optimized validity-preserving application conditions for graph transformation rules, in: E. Guerra, F. Orejas (Eds.), Graph Transformation, Springer International Publishing, Cham, 2019, pp. 177–194.

[22] No Magic, Magic draw, https://www.nomagic.com/products/magicdraw.

[23] T. Kehrer, U. Kelter, G. Taentzer, Consistency-preserving edit scripts in model versioning, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, IEEE, Piscataway, 2013, pp. 191–201.

[24] S. Lack, P. Sobociński, Adhesive and quasiadhesive categories, Theor. Inform. Appl. 39 (3) (2005) 511–545, https://doi.org/10.1051/ita:2005028.

[25] M. Löwe, H. König, C. Schulz, M. Schultchen, Algebraic graph transformations with inheritance and abstraction, Sci. Comput. Program. 107–108 (2015) 2–18, https://doi.org/10.1016/j.scico.2015.02.004, selected Papers from the Brazilian Symposiums on Formal Methods (SBMF 2012 and 2013).

[26] N. Nassar, H. Radke, T. Arendt, Rule-based repair of EMF models: an automated interactive approach, in: E. Guerra, M. van den Brand (Eds.), Theory and Practice of Model Transformation – 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10374, Springer, 2017, pp. 171–181.

[27] H. Ehrig, U. Golas, A. Habel, L. Lambers, F. Orejas, $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation, Math. Struct. Comput. Sci. 24 (4) (2014), https://doi.org/10.1017/S0960129512000357.

[28] R. Heckel, A. Wagner, Ensuring consistency of conditional graph grammars, Electron. Notes Theor. Comput. Sci. 2 (Supplement C) (1995) 118–126, https://doi.org/10.1016/S1571-0661(05)80188-4.

[29] A. Schürr, Specification of graph translators with triple graph grammars, in: E.W. Mayr, G. Schmidt, G. Tinhofer (Eds.), Graph-Theoretic Concepts in Computer Science, in: Lecture Notes in Computer Science, vol. 903, Springer, 1995, pp. 151–163.

[30] OMG, OMG unified modeling language. Version 2.5, http://www.omg.org/spec/UML/2.5/, 2015.

[31] Eclipse Foundation, Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/, 2019.

[32] OCLinEcore, Eclipse OCL, https://wiki.eclipse.org/OCL/OCLinEcore, 2019.

[33] OCL, Eclipse OCL, https://projects.eclipse.org/projects/modeling.mdt.ocl, 2019.

[34] Z. Ujhelyi, G. Bergmann, A. Hegedüs, A. Horváth, B. Izsó, I. Ráth, Z. Szatmári, D. Varró, EMF-IncQuery, Sci. Comput. Program. 98 (P1) (2015) 80–99, https://doi.org/10.1016/j.scico.2014.01.004.

[35] N. Nassar, J. Kosiol, T. Kehrer, G. Taentzer, Generating large EMF models efficiently - a rule-based, configurable approach, in: H. Wehrheim, J. Cabot (Eds.), Fundamental Approaches to Software Engineering, Springer International Publishing, Cham, 2020, pp. 224–244.

[36] N. Behr, M.G. Saadat, R. Heckel, Commutators for stochastic rewriting systems: theory and implementation in Z3, arXiv:2003.11010, 2020.

[37] A. Corradini, T. Heindel, F. Hermann, B. König, Sesqui-pushout rewriting, in: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (Eds.), Graph Transformations, Springer, Berlin and Heidelberg, 2006, pp. 30–45.

[38] K.-H. Pennemann, Development of Correct Graph Transformation Systems, Ph.D. thesis, Carl von Ossietzky-Universität Oldenburg, 2009, https://nbn-resolving.org/urn:nbn:de:gbv:715-oops-9483.
[39] M. Giese, D. Larsson, Simplifying transformations of OCL constraints, in: L. Briand, C. Williams (Eds.), Model Driven Engineering Languages and Systems, Springer, Berlin and Heidelberg, 2005, pp. 309–323.
[40] J.S. Cuadrado, Optimising OCL synthesized code, in: A. Pierantonio, S. Trujillo (Eds.), Modelling Foundations and Applications, Springer International Publishing, Cham, 2018, pp. 28–45.
[41] K. Azab, A. Habel, K.-H. Pennemann, C. Zuckschwerdt, ENFORCe: a system for ensuring formal correctness of high-level programs, in: Proc. 3rd International Workshop on Graph Based Tools, GraBaTs'06, vol. 1, 2006, pp. 82–93.
[42] R. Clarisó, J. Cabot, E. Guerra, J. de Lara, Backwards reasoning for model transformations: method and applications, J. Syst. Softw. 116 (Supplement C) (2016) 113–132, https://doi.org/10.1016/j.jss.2015.08.017.
[43] J. Dyck, H. Giese, Inductive invariant checking with partial negative application conditions, in: F. Parisi-Presicce, B. Westfechtel (Eds.), Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings, in: Lecture Notes in Computer Science, vol. 9151, Springer, 2015, pp. 237–253.
[44] J. Dyck, H. Giese, k-inductive invariant checking for graph transformation systems, in: J. de Lara, D. Plump (Eds.), Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF, 2017, Marburg, Germany, July 18-19, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10373, Springer, 2017, pp. 142–158.
[45] B. Becker, D. Beyer, H. Giese, F. Klein, D. Schilling, Symbolic invariant verification for systems with dynamic structural adaptation, in: L.J. Osterweil, H.D. Rombach, M.L. Soffa (Eds.), 28th International Conference on Software Engineering, ICSE 2006, Shanghai, China, May 20–28, 2006, ACM, 2006, pp. 72–81.
[46] J. Kosiol, L. Fritsche, N. Nassar, A. Schürr, G. Taentzer, Constructing constraint-preserving interaction schemes in adhesive categories, in: J.L. Fiadeiro, I. Tuțu (Eds.), Recent Trends in Algebraic Development Techniques, Springer International Publishing, Cham, 2019, pp. 139–153.
[47] N. Macedo, T. Jorge, A. Cunha, A feature-based classification of model repair approaches, IEEE Trans. Softw. Eng. 43 (7) (2017) 615–640, https://doi.org/10.1109/TSE.2016.2620145.
[48] C. Nentwich, W. Emmerich, A. Finkelstein, Consistency management with repair actions, in: L.A. Clarke, L. Dillon, W.F. Tichy (Eds.), Proceedings of the 25th International Conference on Software Engineering, May 3–10, 2003, Portland, Oregon, USA, IEEE Computer Society, 2003, pp. 455–464, https://ieeexplore.ieee.org/xpl/conhome/8548/proceeding.
[49] N. Nassar, J. Kosiol, H. Radke, Rule-based repair of emf models: formalization and correctness proof, in: Graph Computation Models (GCM 2017), Electronic Pre-Proceedings, 2017, http://pages.di.unipi.it/corradini/Workshops/GCM2017/papers/Nassar-Kosiol-Radke-GCM2017.pdf.
[50] G. Taentzer, M. Ohrndorf, Y. Lamo, A. Rutle, Change-preserving model repair, in: M. Huisman, J. Rubin (Eds.), Fundamental Approaches to Software Engineering – 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10202, Springer, 2017, pp. 283–299.
[51] M. Ohrndorf, C. Pietsch, U. Kelter, T. Kehrer, Revision: a tool for history-based model repair recommendations, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, 2018, pp. 105–108.
[52] A. Habel, C. Sandmann, Graph repair by graph programs, in: M. Mazzara, I. Ober, G. Salaün (Eds.), Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25–29, 2018, in: Lecture Notes in Computer Science, vol. 11176, Springer, 2018, pp. 431–446, Revised Selected Papers.
[53] C. Sandmann, A. Habel, Rule-based graph repair, in: R. Echahed, D. Plump (Eds.), The Tenth International Workshop on Graph Computation Models (GCM 2019) – Proceedings, 2019, pp. 49–64, http://gcm2019.imag.fr/pages/Proceedings-GCM2019.pdf.
[54] S. Sen, B. Baudry, D. Precup, Partial model completion in model driven engineering using constraint logic programming, in: Proc. INAP'07, 2007, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.386.1873.
[55] K.R. Apt, M. Wallace, Constraint Logic Programming Using Eclipse, Cambridge University Press, 2007.
[56] Á. Hegedüs, Á. Horváth, I. Ráth, M.C. Branco, D. Varró, Quick fix generation for dsmls, in: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18–22, 2011, 2011, pp. 17–24.
[57] N. Macedo, T. Guimarães, A. Cunha, Model repair and transformation with echo, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, 2013, pp. 694–697.
[58] S. Schneider, L. Lambers, F. Orejas, A logic-based incremental approach to graph repair, in: R. Hähnle, W.M.P. van der Aalst (Eds.), Fundamental Approaches to Software Engineering – 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, in: Lecture Notes in Computer Science, vol. 11424, Springer, 2019, pp. 151–167.