Constructing Optimized Validity-Preserving Application Conditions for Graph Transformation Rules

Extended Version

Nebras Nassar[1], Jens Kosiol[1], Thorsten Arendt[2], and Gabriele Taentzer[1]

[1] Philipps-Universität Marburg, Marburg, Germany
{nassarn,kosiolje,taentzer}@informatik.uni-marburg.de
[2] thorsten.arendt@uni-marburg.de

Abstract. There is an increasing need for graph transformations ensuring valid result graphs wrt. a given set of constraints. In a model refactoring process, for example, each performed refactoring should yield a valid model graph. At least, it has to remain an element of the underlying modeling language. If a graph transformation rule always produces valid output, it is called *validity-guaranteeing*; if only when applied to an already valid graph, it is called *validity-preserving*. There is a formal construction for graph transformation systems making them validity-guaranteeing. This is ensured by adding a validity-guaranteeing application condition to each of its transformation rules. This theory has been implemented recently as an Eclipse plug-in called OCL2AC. Initial tests have shown that resulting application conditions can become pretty large. As there are interesting application cases where transformations just need to be validity-preserving (such as model refactoring), we started to investigate this case further. The results are optimizing-by-construction techniques for application conditions for transformations that just need to be validity-preserving. All presented optimizations are proven to be correct. Implementing and evaluating them, we found that the complexity of the resulting application conditions is considerably reduced (by factor 7 on average). Moreover, our optimization yields a speedup of rule application by approximately 2.5 times.

Keywords: Graph Transformation · Constraints · Correctness

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 [31]. 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 as, e.g., argued in [8]. But there are scenarios where even intermediate models have to show validity, 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 [3]. (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 [16]. 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 [32]. (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 [17]. (4) When generating code from abstractly specified model transformations, the transformations should be validity-preserving, especially for safety-critical systems [11].

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 [4]. Constraints are typically expressed as (nested) graph constraints [29,13], into which a large and relevant part of OCL [24] can be translated [28]. 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 an application condition. 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 a graph, the resulting graph is valid. As for tool support, OCL2AC [19] automatically translates OCL constraints into graph constraints and integrates these as application conditions into transformation rules specified in Henshin [1]. It computes guaranteeing rules.

Tests of OCL2AC have shown that resulting application conditions can become very complex. Theoretically, application conditions of guaranteeing rules grow over-exponentially in the worst case [26]. As there are interesting application cases where transformations just need to be validity-preserving (as pointed out above), it is worthwhile to investigate validity-preserving transformations further. Habel and Pennemann [13] present a direct construction of the logically weakest application condition, enough to preserve validity. As this kind of condition is logically weaker, our expectation was in the beginning that it can be expressed in a simpler form. In contrast, the resulting application conditions may contain even more elements than the validity-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 validity-guaranteeing application condition. The resulting condition can be inherently difficult to simplify because of the used material implication operator. An example is presented in [20].

Contribution and Structure. Focusing on validity preserving transformations only, we develop optimizing-by-construction techniques to construct application conditions that preserve validity and are considerably less complex than the results of the original construction.

1. In Sect. 4, 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 guaranteeing application conditions but simplifies it by omitting parts that check for antecedent validity, while keeping parts that prevent the introduction of violations. This automatic *approximation* of the preserving application condition is conceptually new and quite general in scope. While some of the simplifications are specific for EMF (Thm. 2), the others (Thm. 1) are proven for graph constraints in general and can be easily lifted to adhesive categories [18]. 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.
2. Practically, we have implemented the techniques on top of OCL2AC (Sect. 5) and compared the application conditions of guaranteeing rules with those of preserving ones. The results show a considerable loss in complexity of application conditions (Sect. 6.1).
3. We provide an application case which shows that validity-preserving transformations are useful in practice. In domain-specific model editing (presented as scenario (2) above), every state of the transformation process has to ensure a basic model validity. The example comprises the 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 of validity-preserving transformations.
   In addition, we have conducted several evaluations that do not specifically test our optimization but the overall approach. We compared the run times of validity checking after a transformation using existing OCL validators (*a posterori approach*) with running a validity-preserving transformation (being enriched with application conditions) with and without optimization (*a priori approach*) (Sect. 6.2). Results show that both approaches are fast in practice. Actually, it is the first time that the usability of OCL2AC, and the implemented approach in general, is investigated.

We start our presentation with the running example in Sect. 2 and recall the formal and technical background in Sect. 3. In an appendix, we present all proofs (Appendix A), an extended example (Appendix B), and more details about the evaluation (Appendix C).

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

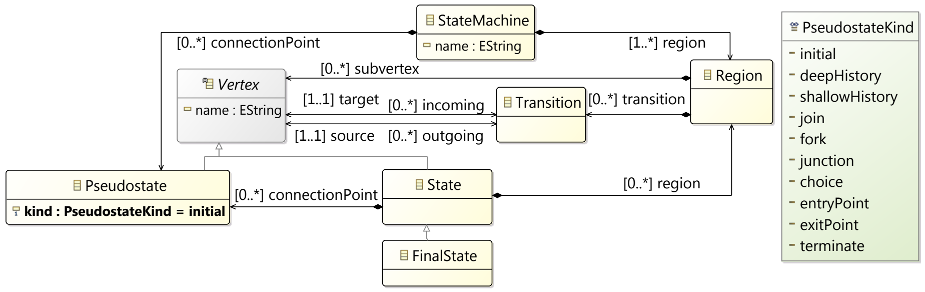
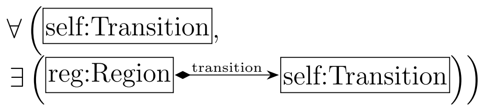


Fig. 1. A simple Statecharts meta-model

the specification of state hierarchies. FinalState inherits from State. Transitions connect Vertices.



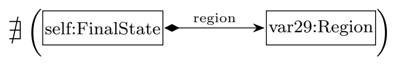


Fig. 2. Graph constraint for TransitionInRegion **Fig. 3.** Graph constraint for no_region

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). Figures 2 and 3 show these constraints as graph constraints, respectively. In the UML, however, these constraints are specified in OCL; the OCL constraint for no_region, for example, is specified as

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

Figure 4 shows a simple transformation rule in Henshin taken from [16] for specifying an edit operation in MagicDraw [21]. 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 [15,16].

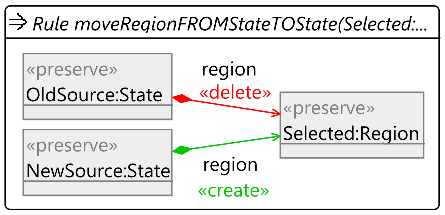


Fig. 4. Transformation rule in Henshin

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 [19], 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 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. Figure 5 presents the resulting guaranteeing application condition which is composed of 7 graphs (explained later in Sect. 4.1). 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.

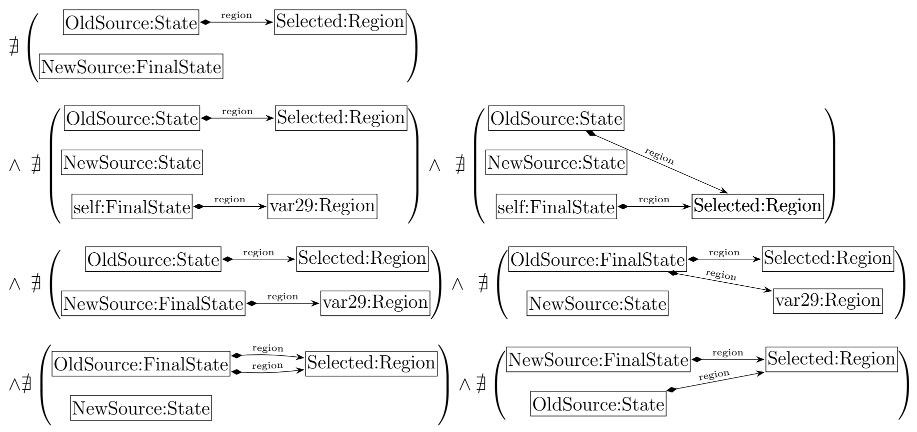


Fig. 5. Non-optimized application condition for moveRegionFromStateToState after integrating the constraint no_region

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

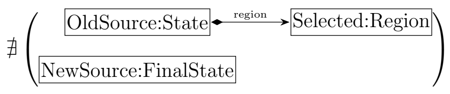


Fig. 6. Optimized application condition for moveRegionFromStateToState still preserving the constraint no_region

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. The guaranteeing condition (not shown), however, consists of three graphs. Thus, assuming valid input models, guaranteeing application conditions can be considerably simplified.

3 Formal Background and Tooling

Our approach is based on the theory of algebraic graph transformation [9]. EMF models and model transformations are formalized as typed attributed graphs and graph transformations as presented in [4]. In the following, we recall (i) *nested graph constraints* and *conditions* as a means to express properties of graphs and graph morphisms and (ii) graph transformation rules as our formal background. Besides, we mention OCL2AC as a tool support.

3.1 Constraints, Conditions, and Rules

Nested graph constraints formulate properties of graphs whereas *nested graph conditions* express properties of graph morphisms [13], i.e., type and structure-preserving mappings between graphs. Graph conditions are mainly used to restrict the applicability of rules. Constraints and conditions are defined recursively as trees of injective morphisms.

Definition 1 (Graph condition). *Given a graph P, a* (nested) graph condition *over P is defined recursively as follows:* `true` *is a graph condition over P and if $a : P \hookrightarrow C$ is an injective morphism and c is a graph condition over C, $(a : P \hookrightarrow C, c)$ is a graph condition over P again. Moreover, Boolean combinations of graph conditions over P are graph conditions over P. A* (nested) graph constraint *is a condition over the empty graph $\varnothing$.*

Satisfaction *of a graph condition d over P for a morphism $p : P \to G$, denoted as $p \models d$, is defined as follows: Every morphism satisfies* `true`*. The morphism p satisfies a condition of the form $d = \exists\,(a : P \hookrightarrow C, c)$ if there exists an injective morphism $q : C \hookrightarrow G$ such that $p = q \circ a$ and q satisfies c. For Boolean operators, satisfaction is defined as usual. A graph G satisfies a graph constraint d, denoted as $G \models d$, if the empty morphism to G does so.*

Graph constraints are expressively equivalent to a first-order logic on graphs [13,29]. To ease notation, we drop the domain of morphisms in constraints and conditions whenever they may be unambiguously inferred and indicate the mapping by the names of nodes. We call constraints of the form $\exists\, C$ *positive* and of the form $\neg\exists\, C$ *negative constraints*. Examples for graph constraints and conditions with informal explanation of their semantics are given in Sect. 2.

Rules are a technical means to declaratively define model transformations.

Definition 2 (Rule. Transformation). *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 graphs L, K, and R, called* left-hand side *(LHS),* interface, *and* right-hand side *(RHS) with two inclusion morphisms $l : K \hookrightarrow L, r : K \hookrightarrow R$. A rule p is* monotonic *if $l : K \hookrightarrow L$ is an isomorphism and* only deletes *if $r : K \hookrightarrow R$ is an isomorphism. The application conditions lac and rac are graph conditions over L and R, respectively.*

Given a rule $\rho = ((L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R), lac, rac)$ *and an injective morphism* $m : L \hookrightarrow G$ *with* $m \models lac$*, called* match*, a* (direct) transformation $G \Rightarrow_{\rho,m} H$ *from* G *to* H *via* ρ *at match* m *is given by the diagram to the right where both squares are pushouts.*

$$\begin{tikzcd}
L \arrow[d, hook, "m \models lac"'] & K \arrow[l, hook', "l"] \arrow[r, hook, "r"'] \arrow[d, hook] & R \arrow[d, hook, "n \models rac"] \\
G & D \arrow[l, hook'] \arrow[r, hook] & H
\end{tikzcd}$$

A rule p *is* applicable *at match* m *if the first pushout square above exists, i.e., if* $m \circ l$ *has a pushout complement* D*, and, moreover, the match morphism* m *satisfies lac and the* co-match n *satisfies rac.*

Note that the first pushout square exists if and only if the match m fulfills the dangling edge check ensuring that a rule application at this match would not let an edge dangle. Applying the rule, the elements of $m(L \backslash K)$ are deleted. Then, at the chosen image of K in G, a copy of $R \backslash K$ is created. Afterwards, the resulting mapping of the graph R into the new graph is checked to fulfill the right application condition of the rule. In that case, the new graph is the result of the rule application.

An example of a rule is shown in Fig. 4. Right application conditions are important in theory but not necessary in practice as they may equivalently be transformed into left application conditions. Therefore, application conditions are understood to be left application conditions.

Computing application conditions from graph constraints. Given a rule and a constraint, one computes all the different ways in which the constraint may be satisfied after applying the rule. This is done by overlapping its RHS in all possible ways with the graphs of the constraint. This computation is iterated along the nesting structure of the constraint. The result is a right application condition for the rule that is satisfied only if the constraint is valid after rule application. By applying the inverse rule to this right application condition, again along its nesting structure, a left application condition is received still *guaranteeing* validity w.r.t. the given constraint. Adding the premise that the constraint was already valid before rule application yields the *preserving application condition.*

Starting in [14] for special cases in the category of graphs, this construction has been generalized to arbitrary nested constraints in the general setting of $\mathcal{M}$-adhesive categories [13].

Fact 1 ([13]). *Given a plain rule* $p = (L \hookleftarrow K \hookrightarrow R)$ *and a graph constraint* c *there are constructions* $\mathsf{Gua}(p, c)$ *and* $\mathsf{Pres}(p, c)$ *equipping* p *with an application condition ac such that* $H \models c$ *for every transformation* $G \Rightarrow_{\mathsf{Gua}(p,c)} H$ *and* $H \models c$ *for every transformation* $G \Rightarrow_{\mathsf{Pres}(p,c)} H$ *where* $G \models c$*.*

3.2 OCL2AC Tool

OCL2AC [19] is an Eclipse plug-in implementing the existing theory [13,28] for adapting a given rule-based model transformation such that resulting models guarantee a given constraint set. OCL2AC consists of two main components: (1) OCL2GC takes a meta-model [7] and a set of OCL constraints as inputs and automatically returns a set of semantically equivalent graph constraints as output. (2) GC2AC takes a transformation rule defined in Henshin and a graph constraint and automatically returns the Henshin rule with an updated application condition guaranteeing the given graph constraint. Each component can be used independently as an Eclipse-based tool.

Limitations. The general formal approach we are based on, and hence OCL2AC as well, come 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.

4 Optimizing Application Conditions

The application conditions being calculated by the approach of the tool OCL2AC guarantee validity even if the input is not a valid EMF-model. Since we focus on validity preservation of EMF-models in this paper, the calculated conditions can be considerably simplified. In this section, we investigate several strategies to construct optimized validity-preserving application conditions.

4.1 Approximating Preservation

In common application scenarios (like refactoring), a user can assume that rules are applied to instances showing a certain validity. Hence, when applying a rule, an already valid constraint does not need to be guaranteed but just preserved. The construction Pres of a preserving rule (as mentioned in Fact 1) takes this into account. Though being logically weaker, the resulting application condition can be even more complex with respect to the structure and number of contained graphs and simplification is inherently difficult. Nevertheless, it is possible to simplify guaranteeing application conditions *during the construction process* if they just need to preserve validity. In the following, we present three forms of simplification.

1. We collect all rule elements being deleted or created and check if this set overlaps with the set of all constraint elements. If this overlap is empty, the resulting preserving 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 optimized application condition is just true.
3. When calculating an application condition, a constraint graph is overlapped with the RHS graph of a 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 the application conditions arising from those cases would just check that the input graph already fulfills the constraint.

Especially the third simplification omits cases where the arising graph in the application condition contains nodes not connected to nodes of the LHS of the rule, thus amounting to global checks upon application. We state the correctness of these simplifications in the following theorem.

Theorem 1 (Correctness of simplifications). *Let c be a graph constraint and $p = (L \hookleftarrow K \hookrightarrow R)$ be a plain rule. Let $\rho = (p, ac)$ be the same plain rule equipped with the application condition ac computed in one of the following ways:*

1. *If both the elements of $L \backslash K$ and the elements of $R \backslash K$ intersect emptily with every graph C occurring in the constraint c, then $ac =$ `true`.*
2. *If p is monotonic and c is a positive constraint, then $ac =$ `true`. Analogously, if p only deletes and c is a negative constraint, then $ac =$ `true`.*
3. *If $c = \neg\exists\, C$, let $Gua(p, c)$ yield the right application condition $rac := \neg(\bigvee_{i\in I} \exists\, P_i)$ with morphisms $c_i : C \hookrightarrow P_i$ and $r_i : R \hookrightarrow P_i$. Let $rac_{pres} := \neg(\bigvee_{j\in J} \exists\, P_j)$ with $J \subseteq I$ including only those P_i where $c_i(C) \cap r_i(R \backslash K) \neq \emptyset$. Then ac is the application condition that arises by translating the right application condition rac_{pres} to the LHS of rule p.*

Then for all transformations $G \Rightarrow_{\rho=(p,ac)} H$ where $G \models c$ also $H \models c$.

The proof follows a common pattern in all cases: Checking for the (non-)existence of graphs occurring in the constraint in all these cases is *sequentially independent* from application of the rule. Hence, checking the constraint for validity always gives the same result, no matter if done before or after rule application.

Example 1 (compare Sect. 2). Constraint no_region is required to be integrated into rule moveRegionFromStateToState since a region-edge is created by this rule and contained in this constraint. Figure 5 shows the guaranteeing application condition. The first graph (the uppermost graph) results from a maximal overlapping of the constraint with the rule. Note that it is possible to identify nodes of types State and FinalState 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 forth 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. In every case, the overlapping of the constraint with the RHS is then translated to the LHS of the rule.

Our proposed optimizations lead to the result displayed in Fig. 6 by the application of Thm. 1, 3.: Except for the subcondition containing the uppermost graph, all other subconditions in Fig. 5 are omitted. The uppermost one has to be saved because the region-edge created by the rule is overlapped 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 leading to global checks. To conclude, only one local check remains.

Example 2 (compare Sect. 2). The constraint TransitionInRegion is not required to be integrated into the rule moveRegionFromStateToState. Thm. 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 that occur in the constraint.

4.2 Dealing with EMF's Built-in Negative Constraints

EMF has several built-in constraints [4]. Instance models that do not satisfy these EMF-constraints cannot even be opened in the EMF-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 $\exists\, A$ with A 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 actual guaranteeing rule 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 $p : P \hookrightarrow G$ 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'.*

Correctness of this theorem is proven by 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.

Example 3 (compare Sect. 2). Thm. 2 would drop the third, sixth, and seventh subcondition from the application condition in Fig. 5 by replacing it with false since it contains a node with more than one container or parallel edges.

5 Tooling

We developed our optimizer as an Eclipse-plugin tool support on top of OCL2AC implementing all of the proposed simplifications except for the elimination of containment cycles. The optimizer consists of two main components: (a) an *analyzer* that detects if a constraint needs to be integrated into a given rule at all (Thm. 1, 1 and 2) and (b) a *simplifier* for eliminating unnecessary subconditions from the guaranteeing application conditions *during the construction process* (Thm. 1, 3 and Thm. 2). Given a Henshin rule and a graph constraint, our optimizer automatically renders the rule to preserve the validity of the constraint. Additionally, we implemented simplifications of application conditions by applying well-known equivalence rules like $\exists (C_1, \exists C_2) \equiv \exists C_2$ if $C_1 \subseteq C_2$, $\exists C_1 \vee \exists C_2 \equiv \exists C_1$ if $C_1 \subseteq C_2$, or $\exists C_1 \wedge \exists C_2 \equiv \exists C_2$ if $C_1 \subseteq C_2$ [26]. Applying these, entire graphs may be omitted and even levels of nesting may be collapsed. The tool support can be downloaded from our website [3].

6 Evaluation

In this section, we show the highlights of our evaluation; a comprehensive overview is given in [20] and the artifacts can be downloaded [3].

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 [16], 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 [25]. Out of 17 original constraints they identified 11 to be enforced in MagicDraw [21]. 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.

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

6.1 Evaluating Complexity

In theory, the size of a computed application condition (the number of graphs) can grow over-exponentially in the worst case compard to the size of the original constraint [26]. 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 1 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. Thm. 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 1. Number of graphs of application conditions and deepest nesting levels before and after optimization (with emphasis on extreme cases)

	w/o optimization		**w optimization**		
Rule	#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

Table 1 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, the number of the resulting graphs is reduced by factor 1.4 only (using Thm. 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 [7] configured to employ the OCLinEcore validator [23] to validate OCL constraints and (b) the OCL interpreter [22]). 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.

Table 2. 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

		Model size						
Scenario	**(Caching)**	**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

Experiment results. Table 2 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 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.

7 Related Work

Related works can be distinguished into two groups: (1) other works ensuring transformation rules to be validity-preserving and (2) simplifying (application) conditions and constraints.

Ensuring transformation rules to be validity-preserving. In [2,27], Azab, Pennemann et al. introduce ENFORCe, a prototype implementation that can ensure the correctness of graph programs. It integrates graph constraints as left application conditions of rules as well but supports (partially) labeled graphs, not EMF models, and there is no translation from OCL to graph constraints available.

Clarisó et al. present in [5] 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. Resulting application conditions are not further optimized, neither by ENFORCe nor in the work by Clarisó et al. To the best of our knowledge, our work is the only one which optimizes the resulting application conditions considerably.

Simplifying (application) conditions and constraints. Rules for semantic equivalences in graph constraints and conditions have been reported in several places [26,27,28] 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 [12,6]. Depending on the usage scenario, such simplifications could provide a valuable pre-processing step to our approach.

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 in [13] yields validity-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 Thm. 1 and Thm. 2) to construct optimized validity-preserving application conditions and implemented them on top of OCL2AC. In our evaluation, the usability of OCL2AC was investigated for the first time, 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 and by checking negative invariants of modeling languages. Our ultimate goal is to obtain understandable application conditions identifying exactly those portions of the given constraints that are relevant for a given rule. 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 [27,30].

Acknowledgement. We are grateful to Annegret Habel, Christian Sandmann, and Steffen Vaupel for their helpful comments on a draft version of this paper. This work was partially funded by the German Research Foundation (DFG), projects “Generating Development Environments for Modeling Languages” and “Triple Graph Grammars (TGG) 2.0”.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Proc. of MODELS 2010. pp. 121–135. Springer, Berlin and Heidelberg (2010)
2. Azab, K., Habel, A., Pennemann, K.H., Zuckschwerdt, C.: ENFORCe: A System for Ensuring Formal Correctness of High-level Programs. In: Proc. 3rd International Workshop on Graph Based Tools (GraBaTs’06). vol. 1, pp. 82–93 (2006)
3. Becker, B., Lambers, L., Dyck, J., Birth, S., Giese, H.: Iterative Development of Consistency-Preserving Rule-Based Refactorings. In: Theory and Practice of Model Transformations. pp. 123–137. Springer, Berlin (2011)
4. Biermann, E., Ermel, C., Taentzer, G.: Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation. Software & Systems Modeling **11**(2), 227–250 (2012)
5. Clarisó, R., Cabot, J., Guerra, E., de Lara, J.: Backwards Reasoning for Model Transformations: Method and Applications. Journal of Systems and Software **116**(Supplement C), 113–132 (2016)
6. Cuadrado, J.S.: Optimising OCL Synthesized Code. In: Modelling Foundations and Applications. pp. 28–45. Springer International Publishing, Cham (2018)
7. Eclipse Foundation: Eclipse Modeling Framework (EMF) (2019), http://www.eclipse.org/emf/

8. Egyed, A.: Instant Consistency Checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering. pp. 381–390. New York (2006)
9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Berlin and Heidelberg and New York (2006)
10. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation – General Framework and Applications. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2015)
11. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards Verified Model Transformations. In: Proc. of the 3rd International Workshop on Model Development, Validation and Verification (MoDeV2a), Genova. pp. 78–93 (2006)
12. Giese, M., Larsson, D.: Simplifying Transformations of OCL Constraints. In: Model Driven Engineering Languages and Systems. pp. 309–323. Springer, Berlin and Heidelberg (2005)
13. Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. Mathematical Structures in Computer Science **19**, 245–296 (2009)
14. Heckel, R., Wagner, A.: Ensuring Consistency of Conditional Graph Grammars. Electronic Notes in Theoretical Computer Science **2**(Supplement C), 118–126 (1995)
15. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-Preserving Edit Scripts in Model Versioning. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013. pp. 191–201. IEEE, Piscataway (2013)
16. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically Deriving the Specification of Model Editing Operations from Meta-Models. In: Theory and Practice of Model Transformations (ICMT 2016). pp. 173–188. Springer, Heidelberg (2016)
17. Krause, C., Giese, H.: Probabilistic Graph Transformation Systems. In: Graph Transformations. pp. 311–325. Springer, Berlin and Heidelberg (2012)
18. Lack, S., Sobociński, P.: Adhesive and Quasiadhesive Categories. Theoretical Informatics and Applications **39**(3), 511–545 (2005)
19. Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: OCL2AC. Automatic Translation of OCL Constraints to Graph Constraints and Application Conditions for Transformation Rules. In: Proc. of ICGT 2018. pp. 171–177. Springer, Cham (2018)
20. Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: Constructing Optimized Validity-Preserving Application Conditions for Graph Transformation Rules: Extended Version. Technical report, Philipps-Universität Marburg (2019), https://uni-marburg.de/fb12/arbeitsgruppen/swt/forschung/publikationen/2019/NKAT19-TR.pdf/
21. No Magic: Magic draw, https://www.nomagic.com/products/magicdraw
22. OCL: Eclipse OCL (2019), https://projects.eclipse.org/projects/modeling.mdt.ocl
23. OCLinEcore: Eclipse OCL (2019), https://wiki.eclipse.org/OCL/OCLinEcore
24. OMG: Object Constraint Language. (2014), http://www.omg.org/spec/OCL/
25. OMG: OMG Unified Modeling Language. Version 2.5 (2015), http://www.omg.org/spec/UML/2.5/
26. Pennemann, K.H.: Generalized Constraints and Application Conditions for Graph Transformation Systems. Diplomarbeit, Department für Informatik, Universität Oldenburg (2004), https://bit.ly/2T4RV0A
27. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Carl von Ossietzky-Universität Oldenburg (2009)
28. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating Essential OCL Invariants to Nested Graph Constraints for Generating Instances of Metamodels. Science of Computer Programming **152**, 38 – 62 (2018)
29. Rensink, A.: Representing First-Order Logic Using Graphs. In: Graph Transformations. pp. 319–335. Springer, Berlin (2004)

30. Schneider, S., Lambers, L., Orejas, F.: Automated reasoning for attributed graph properties. Intl. Journal on Software Tools for Technology Transfer **20**(6), 705–737 (2018)
31. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20**(5), 42–45 (2003)
32. Steimann, F., Frenkel, M., Voelter, M.: Robust Projectional Editing. In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. pp. 79–90. SLE 2017, ACM, New York (2017)

A Proofs

This section contains the proofs of the theorems presented in the main paper.

Proof (of Thm. 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 $p \models$`true` for every injective morphism p.

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 $p : P \rightarrow G$, where G is any EMF-model graph

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

Secondly, if C contains parallel edges or multiple incoming containment edges to the same node, then $c' =$ `false`. Therefore, no injective morphism $p : P \hookrightarrow G$ 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 $q : C \rightarrow G$ into any EMF-model graph G. Hence, there does never exist an injective morphism q s.t. $q \circ a = p$ and $q \models d'$. In summary, $p \not\models c \Leftrightarrow p \not\models c'$ for all injective morphisms $p : P \hookrightarrow G$ for any EMF-model graph G.

The induction steps for Boolean operators are routine. □

The next proof, for all three situations, relies on the notion of parallel independence which we first shortly recall.

Definition 3 (Parallel independence). *Given two plain rules $p_i = (L_i \xhookleftarrow{l_i} K_i \xhookrightarrow{r_i} R_i)$ with $i = 1, 2$, two direct transformations $G \Rightarrow_{p_1,m_1} H_1$ and $G \Rightarrow_{p_2,m_2} H_2$ via those rules are* parallelly independent *if there exist two morphisms $d_1 : L_1 \rightarrow D_2$ and $d_2 : L_2 \rightarrow D_1$ as depicted below such that $m_1 = f_2 \circ d_1$ and $m_2 = f_1 \circ d_2$. The rules p_1 and p_2 are* parallel independent *if every pair of transformations $H_1 \underset{p_1}{\Leftarrow} G \underset{p_2}{\Rightarrow} H_2$ is.*

$$\begin{tikzcd}
R_1 \arrow[d, hook, "n_1"'] & K_1 \arrow[l, hook', "r_1"'] \arrow[r, hook, "l_1"] \arrow[d, hook] & L_1 \arrow[dr, "m_1"'] \arrow[drrr, dashed, "d_1"] & & L_2 \arrow[dl, "m_2"] \arrow[dlll, dashed, "d_2"'] & K_2 \arrow[l, hook', "l_2"'] \arrow[r, hook, "r_2"] \arrow[d, hook] & R_2 \arrow[d, hook, "n_2"] \\
H_1 & D_1 \arrow[l, hook'] \arrow[rr, hook, "f_1"'] & & G & & D_2 \arrow[ll, hook', "f_2"] \arrow[r, hook] & H_2
\end{tikzcd}$$

Proof (of Thm. 1).

1. Let $G \Rightarrow_p H$ be a transformation via rule p where $G \models c$. By induction, we show the stronger statement that for every condition c over any graph P, there exists an injective morphism $g : P \hookrightarrow G$ with $g \models c$ if and only if there exists an injective morphism $g' : P \hookrightarrow H$ with $g' \models c$. In particular, for constraints $c = \exists\,(\varnothing \hookrightarrow C, d)$ this implies that $H \models c$ (via the empty morphism) if and only if $G \models c$. We prove the statement by induction over the structure of the condition. If $c = \texttt{true}$, $H \models c \Leftrightarrow G \models c$.
   Let $c = \exists\,(a : P \hookrightarrow C, d)$, $g : P \hookrightarrow G$ and $g \models c$. By definition, there exists an injective morphism $q : C \hookrightarrow G$ such that $q \circ a = g$ and $g \models d$. By induction hypothesis, there exists an injective morphism $q' : C \hookrightarrow H$ such that $q' \models d$. Consider the constant rule $id_P : P \hookleftarrow P \hookrightarrow P$ (which just checks for the existence of the graph P). The intersection between $L \backslash K$ and P is empty and therefore the rules p and id_P are parallelly independent. In particular, pairs of transformations $G\,{}_{id_P}\!\Leftarrow G \Rightarrow_p H$ are. Hence, for the match morphism g for rule id_P in G, there exists an according match morphism $g' : P \hookrightarrow H$ such that $g' = q' \circ a$ (compare the Local Church-Rosser Theorem and its proof [9]). Thus, $g' \models c$.
   In the other direction, let $g' : P \hookrightarrow H$ and $g' \models c$. By definition, there exists an injective morphism $q' : C \hookrightarrow H$ such that $q' \circ a = g'$ and $q' \models d$. By induction hypothesis, there exists an injective morphism $q : C \hookrightarrow G$ such that $q \models d$. Consider, again, the constant rule $id_P : P \hookleftarrow P \hookrightarrow P$. The intersection between $R \backslash K$ and P is empty and therefore the rules p and id_P are sequentially independent. In particular, transformation sequences $G \Rightarrow_{id_P} G \Rightarrow_p H$ are. This is equivalent to p^{-1} and id_P being parallelly independent. Then, again by the Local Church-Rosser Theorem, there is an injective morphism $g : P \hookrightarrow G$ such that $g = q \circ a$. Thus, $g \models c$.
   Induction over the Boolean operators is standard, again.
2. The proof for this second situation uses parallel independence in a similar fashion; since the considered constraints are not nested, we do not need an induction. Let H denote a graph which arises by application of a monotonic rule $r : K \hookrightarrow R$ to a graph G, i.e., $G \Rightarrow_r H$. A monotonic rule is parallel independent from the rule which just checks for the existence of the graph C, i.e., each pair of transformations $G\,{}_{id_C}\!\Leftarrow G \Rightarrow_r H$ is, where $id_C : C \hookleftarrow C \hookrightarrow C$ is constant. Hence, every match for the graph C in G extends to a match for graph C in H, i.e., $G \models c \Rightarrow H \models c$.
   Dually, checking the existence of a graph C is sequentially independent of applying a rule $l : L \hookleftarrow K$ which only deletes. Hence, every match for C in a

graph H can be extended to a match for C in G, i.e., $H \models \exists\, C \Rightarrow G \models \exists\, C$ and hence $G \models c \Rightarrow H \models c$.

3. Let I be a set indexing the possible overlappings of C and R, i.e., the pairs of jointly surjective, injective morphisms $i_R : R \hookrightarrow P_i, i_C : C \hookrightarrow P_i$. Let $J \subset I$ be the subset of I that consists only of that indices denoting overlappings where at least one newly created element from the rule, i.e., an element from $R \backslash K$ is overlapped with one from C. Let $rac = \neg(\bigvee_{j \in J} \exists\, j_R : R \hookrightarrow P_j)$ be the right application condition of p arising by only considering overlappings from J (and not from the whole set I). Let ac denote the application condition for rule p when moving rac equivalently to the LHS of p.

   We prove the statement by contraction. Thus, let $G \models c = \neg\exists\, C, G \Rightarrow_{\rho=(p,ac),m} H$, so p is equipped with application condition ac, and assume $H \models \neg c = \exists\, C$. This means, there is an injective morphism $q' : C \hookrightarrow H$. By construction of I, there exists an $i \in I$ such that there is an injective morphism $\tilde{q'} : P_i \hookrightarrow H$ with $n = \tilde{q'} \circ i_R$ and $\tilde{q'} \circ i_C = q$ (compare Fig. 7). Since p is equipped with application condition ac, $i \in I \backslash J$; otherwise rule application would have been prevented by that application condition:

$$\begin{aligned} n \not\models \neg\exists\,(i_R : R \hookrightarrow P_i) &\Leftrightarrow n \models \exists\,(i_R : R \hookrightarrow P_i) \\ &\Leftrightarrow \exists\, \tilde{q'} : P_i \hookrightarrow H, \text{ s.t. } n = \tilde{q'} \circ i_R \ . \end{aligned}$$

   Now, like in the proofs above, checking for the existence of $P_i, i \in I \backslash J$ in graph H is independent of applying rule p. Hence, the morphism $\tilde{q'} : P_i \hookrightarrow H$ restricts to an injective morphism $\tilde{q} : P_i \hookrightarrow G$ and therefore $G \models \neg c = \exists\, C$ via the injective morphism $q = \tilde{q} \circ i_C$. This contradicts $G \models c$.

□

$$\begin{tikzcd}
 & & & C \arrow[d, hook, "i_C"] \\
L \arrow[d, hook, "m"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] & R \arrow[r, hook, "i_R"] \arrow[d, hook, "n"] & P_i \arrow[dl, hook', "\tilde{q'}"] \\
G & D \arrow[l, hook', "d_1"] \arrow[r, hook, "d_2"'] & H &
\end{tikzcd}$$

Fig. 7. Existence of satisfying morphism

B C-Preserving Application Conditions as Defined in the Theory [13]: An Example

Using our running example (See Sect. 2), we present the resulting c-preserving application condition ac_{pres} w.r.t. the construction provided by Habel and Pennemann in ([13], Def. 9). The construction of a c-preserving application condition is defined by them using an implication operator as follows:

$$ac_{pres} = (Shift(r^{-1}, c) \Rightarrow ac_{gua}) \quad (1)$$

where r^{-1} denotes the inverse rule of the given rule r, c denotes the given constraint and ac_{gua} denotes the c-guaranteeing application condition defined as $Left(r, (Shift(r, c))$ which is the output (of the second component) of OCL2AC.

Figure 8 presents the c-preserving application condition of integrating the constraint `no_region` into the rule `moveRegionFromStateTOState` being constructed according to the formula (1). The left column displays the antecedent that expresses that the model was already valid before rule application and the right column displays the consequent of the conditional. The resulting application condition ac_{pres} contains 14 graphs although we have integrated only one constraint into the rule. To further simplify the result, the material implication has to be simplified by applying De Morgan's Law and the distributivity law yielding 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—a problem that in NP-complete in general. 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. Our optimizing-by-construction technique simplifies the resulting application conditions *throughout the construction process* without the need for such costly computations. In the best case, as, e.g., in Fig. 6, we even receive the logically weakest possible application condition in a simplified version as output.

C Evaluation Artifacts and Results

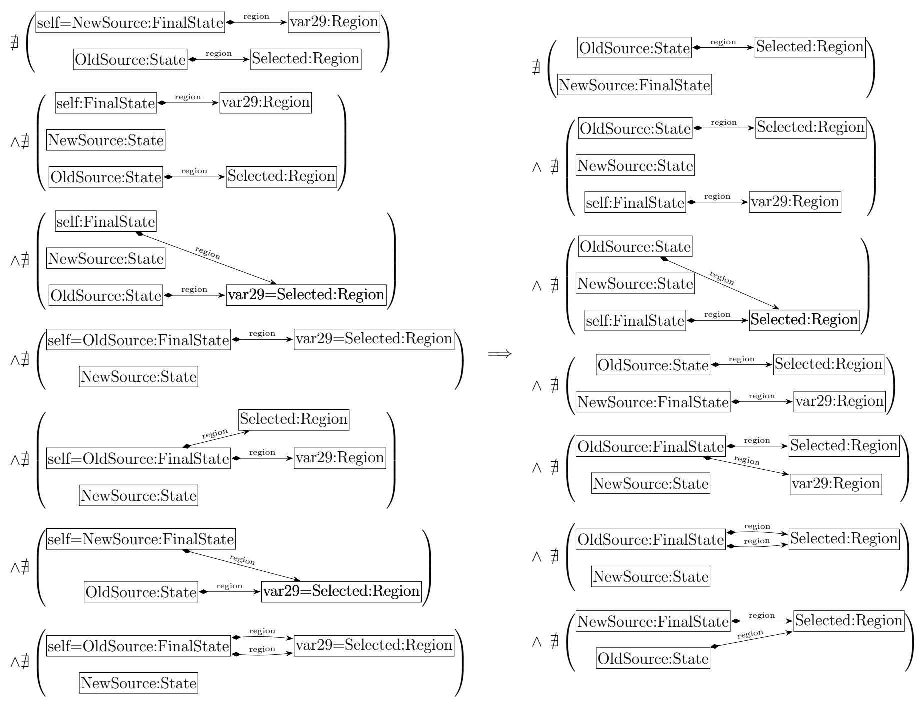


Fig. 8. The resulting c-preserving application condition ac_{pres} w.r.t. the construction provided by Habel and Pennemann in [13], compare with our optimized version in Fig. 6

This section presents the editing rules of the Statecharts (See Table 3) and the 11 OCL constraints (See Table 4) used in the performance experiment. Table 5 presents the number of the resulting graphs, when integrating *without and with optimization* all constraints as application conditions into each rule of the selected 15 rules and the number of the overlapped constraints as well [4].

Table 3. 15 edit rules of all kinds

Rule ID	**Rule Name**
Rule 01	addToConnectionPointReference__entry__Pseudostate
Rule 02	addToStateMachine__submachineState__State
Rule 03	createBehavior__IN__State
Rule 04	createFinalState__IN__Region
Rule 05	createRegion__IN__StateMachine
Rule 06	deleteFinalState__IN__Region
Rule 07	deleteState__IN__Region
Rule 08	deleteTrigger__IN__Transition
Rule 09	moveConstraint__FROM__Transition__guard__TO__Transition__Transition
Rule 10	moveFinalState__FROM__Region__subvertex__TO__Region__Region
Rule 11	moveTransition__FROM__Region__transition__TO__Region__Region
Rule 12	removeFromConnectionPointReference__entry__Pseudostate
Rule 13	removeFromConnectionPointReference__exit__Pseudostate
Rule 14	setState__submachine__TO__StateMachine
Rule 15	unsetState__submachine__TO__StateMachine

[4] For a rule which already has an application condition, the number of its graphs is subtracted from the result.

Table 4. OCL Constraints of Statecharts

ID	OCL Constraints of Statecharts
1	invariant owned: (stateMachine -> notEmpty() implies state -> isEmpty()) and(state -> notEmpty() implies stateMachine -> isEmpty());
2	invariant submachine_states: isSubmachineState=false implies connection -> notEmpty();
3	invariant destinations_or_sources_of_transitions: self.isSubmachineState=true implies (self.connection -> forAll(cp |cp.entry -> forAll(p|p.stateMachine = self.submachine) and cp.exit -> forAll (p | p.stateMachine = self.submachine)));
4	invariant submachine_or_regions: self.isComposite=true implies not (self.isSubmachineState=true);
5	invariant composite_states: self.connectionPoint -> notEmpty() implies self.isComposite=true;
6	invariant no_outgoing_transitions: self.outgoing -> size() = 0;
7	invariant no_regions: self.region -> size() = 0;
8	invariant no_entry_behavior: self.enty -> isEmpty();
9	invariant no_exist_behavior: self.exit -> isEmpty();
10	invariant cannot_reference_submachine: self.submachine -> isEmpty();
11	invariant no_state_behavior: self.doActivity -> isEmpty();

Table 5. Numbers of graphs of application conditions and deepest nesting levels before and after optimization for the selected 15 rules

	w/o optimization		**w optimization**		
Rule	#graphs	level	#graphs	level	#overlapped const.
addToConnectionPoint..	58	6	30	6	(1)
addToStateMachine..	77	6	25	6	(2)
createBehavior..	70	6	1	1	(3)
createFinalState..	44	6	31	6	(11)
createRegion..	41	6	7	2	(2)
deleteFinalState..	42	6	29	6	(11)
deleteState..	42	6	29	6	(11)
deleteTrigger..	37	6	0	0	(0)
moveConstraint..	39	6	0	0	(0)
moveFinalState..	86	6	0	0	(0)
moveTransition..	49	6	0	0	(0)
removeEntry..	55	6	27	6	(1)
removeExit..	55	6	27	6	(1)
setState_submachine..	77	6	25	6	(2)
unsetState_submachine..	74	6	22	6	(2)
Average (15 rules)	56.4	6	16.8	3.8	3.1