Automatically Deriving the Specification of Model Editing Operations from Meta-Models

Timo Kehrer[1,3], Gabriele Taentzer[2], Michaela Rindt[3], Udo Kelter[3]

[1]Politecnico di Milano, Italy
[2]Philipps-Universität Marburg, Germany
[3]University of Siegen, Germany

Abstract To optimally support continuous model evolution in model-based software development, adequate tool support for model version management is needed. Instead of reporting model differences to the developer line-by-line or element-wise, their grouping into semantically associated change sets helps in understanding model differences. Edit operations are the concept of choice to group such change sets. Considering visual models in particular, edit operations preserve a basic form of consistency such that changed models can still be viewed in a standard editor. Using edit operations for the version management of domain-specific models requires tool developers to specify all necessary edit operations in order to produce or replicate every possible change on a model. However, edit operations can be numerous and their manual specification is therefore tedious and error-prone. In this paper, we present a precise approach to specify a complete set of consistency-preserving edit operations for a given modeling language. The approach is supported by a generator and has been evaluated in four case studies covering several visual modeling languages and standard editors.

Keywords: Model-driven engineering, model consistency, model editing, meta-model

1 Introduction

Model-driven engineering (MDE) raises the level of abstraction in engineering by using models as primary development artifacts. In particular, domain-specific modeling languages (DSMLs) promise to increase productivity and quality of developments. The increase of productivity highly depends on the quality of the provided tool environment, which has to be customized to the DSML.

To optimally support model evolution, developers need adequate tools for model versioning tasks, including comparison, patching, and merging of models. Currently available tools mostly display and operate with low-level model changes which assume a textual or graph-based internal model representation. Such low-level changes are hard to understand for average tool users and often confusing [2]. Moreover, patching and merging those low-level changes may lead to inconsistent models [19]. Version management of visual models may trap into particular pitfalls: It can happen that the synthesized result model can no longer be opened in visual editors and must be corrected based on a serialized data format (e.g. XML) by using textual editors, which is obviously not attractive or even no option at all.

Recent advances in model versioning [20,21] address this problem by lifting model versioning concepts and tools to higher-level edit operations. Edit commands in visual editors are typical forms of edit operations. They are better suited to explain changes or to resolve conflicts since they cluster semantically associated low-level changes and thus raise the abstraction level of model version management. Edit operations are consistency-preserving in the sense that they always lead to model versions that can be further displayed and edited. Therefore, they are a promising solution to the problem that patching and merging can fail at any point of time.

In model editors, specifications of the available edit operations are typically hidden in the tool implementation. However, explicit declarative specifications of edit operations are required as configuration parameter for the calculation of model differences in [20,21]. In-place model transformations are well-suited for that purpose [20,21,22]. In [20], edit operations are specified by model transformation rules, called *edit rules*. A set of edit rules must meet three challenging requirements. To be a suitable basis for model patching and merging, *edit rules must preserve the level of consistency being enforced by the editor*, i.e. synthesized results can always be opened and corrected if needed (R1). In order to obtain model differences which capture the changes between model versions correctly, *a set of edit rules must be complete for a given DSML in the sense that every model modification can be expressed by using rules of this set* (R2). To be understandable by tool users, *edit rules should mimic the behavior of visual editors for the given DSML* (R3). The specification of an edit rule set which meets these requirements is a tedious and error-prone task when done manually.

Figure 1 outlines a methodology to deduce a suitable set of edit rules in a step-wise manner. The meta-model of a given DSML serves as initial input of this process. Such a meta-model is usually *perfect* in the sense that it specifies valid models with well-defined semantics, which can be successfully processed by code generators or model interpreters. The *perfect meta-model* may be standardized or stem from an authority such as a research standardization group or tool vendor. The further processing is based on two general observations. Firstly, many modeling editors do not fully comply with the standard, i.e., certain language features are not supported. Secondly, visual editors usually do not enforce all consistency constraints defined in their DSMLs. These observations apply to, e.g., UML editors such as Magic Draw [23], RSA [16] and EMF-based editors [8]. Thus, the original meta-model is reduced to a meta-model *effectively* used by the editor (Step 1 in Figure 1). For this reduction, parts of the meta-model related to unsupported language features can be deleted. To make the effective notion of consistency explicit, certain multiplicities can be relaxed and unsupported well-formedness rules (typically formulated using the OCL) can be dropped. The obtained *effective meta-model* forms the basis for Step 2, the automated specification of all elementary edit rules. In Step 3, these rules may be further composed to specify more complex edit operations such as refactorings.

In previous work [26] we sketched our ideas and focused on their implementation and tooling. In this paper, we focus on the second step of the workflow outlined in Figure 1. The contributions over previous work are the following: (1) We present an algorithm for generating edit rules from a meta-model with restricted multiplicities, which we claim to be a sufficient degree of consistency for most effective meta-models. (2) We argue that our approach is able to generate a complete set of consistency-preserving edit rules, i.e. it meets requirements R1 and R2. (3) Concerning requirement R3, we show empirically that our approach is meaningful from a practical point of view.

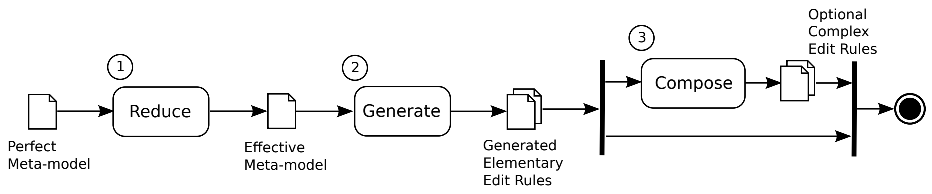

Figure 1. Process for creating a set of consistency-preserving edit operations

The paper is structured as follows: We start with an example in Section 2. The formal basis for this work are graphs and graph transformations, they are recalled in Section 3. The generation of a complete set of consistency-preserving edit rules is presented in Section 4. Our approach is evaluated in Section 5. Sections 6 and 7 present the related work and the conclusion.

2 Running Example

In this section, we informally present how a simplified meta-model for state machines [14,24] is used to generate a complete set of consistency-preserving edit operations. The meta-model is shown in Figure 2. It contains the main model element types of state machines such as *State* and *Transition* as well as inter-relations like *source* and *target*. Moreover, it contains multiplicities requiring, e.g., that each transition must have a source and a target state. In addition, correct state machines have to fulfill further constraints, e.g. transitions are not allowed to connect states of two parallel regions. Usual visual editors can load and edit models which do not satisfy these advanced constraints. Thus we do not consider them here, i.e. the

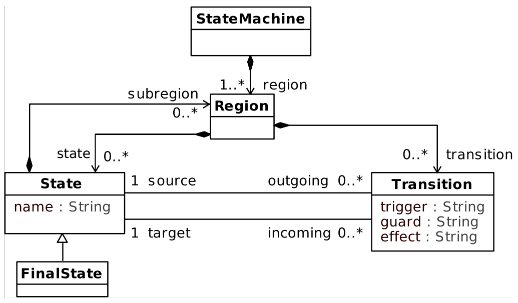

Figure 2. Effective meta-model of simple UML state machines

meta-model in Figure 2 is effective; it can serve as underlying meta-model for all models to be edited, but is less restrictive than the UML standard meta-model for state machines.

A total of 25 edit rules are generated, they are available on the accompanying website of this paper [1]. Due to space limitations, we focus on the creation rules here and neglect all other kinds of rules. A subset of the generated creation rules is illustrated in Figure 3. We present the rules in an integrated form: the left- and right-hand sides of a rule are merged into one graph following the visual syntax of the model transformation language Henshin [3]. The left-hand side of a rule comprises all model elements stereotyped by delete and preserve. The right-hand side contains all model elements annotated by preserve and create.

The following rules are generated: The rule create_StateMachine creates the root node. Since it has a mandatory child of type *Region*, a model element of that type has to be created as well. Moreover, there are rules create_FinalState_state, cre-

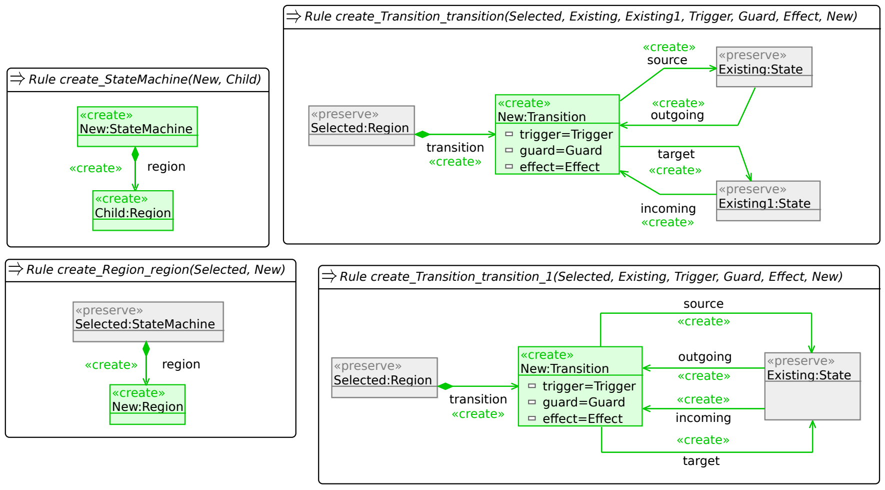

Figure 3. Subset of generated creation rules for UML state machines

ate_State_state, create_Region_region and create_Region_subregion (not shown in Figure 3). The rule create_Transition_transition creates a transition and immediately connects it to its source and target states, which are so-called mandatory neighbors. Since the edge types *source* and *target* are parallel and both have a multiplicity [1..1] (s. Figure 2), we get a second variant of this rule: create_Transition_transition_1. This variant creates edges of types *source* and *target* referencing the same *State* node, i.e. it creates a "loop" in the visual representation. Note that attribute declarations are conceptually handled as special edge types with a *fixed* multiplicity of [1..1]. Thus, attribute values are treated as mandatory neighbors as well. This implies that all attributes have to be set in newly created nodes.

3 Background

The formal underpinning of edit rule generation is based on graphs and graph transformation as presented in [4]. Here, we recall all needed concepts from [4] in a semi-formal way.

3.1 Graph-based Representation of Models

Graphs are a natural means to formally define models and meta-models. While a meta-model defines the allowed types formalized by type graphs, models are considered as instances of meta-models and formally treated as typed graphs. Hence, we abstract from the graphical layout of visual models here and concentrate on the underlying structure. In this sense, we consider models and graphs as synonyms. A graph consists of a set of nodes, a set of edges, each one running from source to target node.

A meta-model is basically a graph containing all type information including a *type hierarchy* to represent the inheritance relation, a set of abstract types, a containment relation between type nodes and a relation of opposite edge types. Moreover, *multiplicities* can be attached to edge types. A multiplicity is a pair $[lb, ub]$ with $lb \leq ub$ or $ub = *$. An edge type et is called *required* if $et.lb > 0$, *bounded* if $et.ub \neq *$, *fixed* if $et.lb = e.ub$, and *many* if $(et.ub > 1)$ or $et.ub = *$. Note that these properties are not mutually exclusive. A node type without incoming containment edge types and without super types having incoming containment edge types is called *root* type. Attributes are usually single-valued, i.e., neither `null`-values nor multiple values are allowed. I.e., a multiplicity of [1..1] is implicitly assigned to each attribute declaration in a type graph.

An edge with containment type is called *containment edge*. Its source and target nodes are referred to as *parent* (or *container*) and *child*, respectively. The target node of a non-containment edge is called a *neighbor* of the respective source node. Target nodes of edge types with multiplicity property *required* are also referred to as *mandatory neighbors* and *mandatory children* [28].

3.2 Consistency of Models

A model M is considered (syntactically) consistent w.r.t. a meta-model MM if it is properly typed over MM and if it meets the consistency constraints specified in MM. We distinguish among *basic consistency constraints*, *multiplicity invariants* and further *well-formedness rules.*

Basic consistency constraints correspond to fundamental conditions imposed by EMOF-based modeling frameworks. A formal treatment of basic consistency constraints can be found in [4]; they can be summarized as follows: (1) The model graph is *correctly typed* w.r.t. a given type graph deduced from a meta-model. (2) Each node has *at most one container* and *cycles of containment edges do not occur.* (3) There are *no parallel edges* of the same type. Edges are parallel if they have the same source and target node. (4) For *all pairs of opposite edge types* (et_1, et_2): If there is an edge of type et_1 then there is also an edge of type et_2 linking the same nodes in the opposite direction, and vice versa.

3.3 Specification of Edit Operations

In our approach, we use in-place model transformation techniques which are based on graph transformation concepts [10]. This enables us to precisely specify edit operations as declarative transformation rules which we call *edit rules*. An edit rule specifies *i)* the *conditions* under which the rule is applicable and *ii)* a set of *change actions* which are to be performed when the rule is applied. Each change action corresponds to a primitive graph operation, i.e., the creation/deletion of a model element or the setting of an attribute value.

A *rule* $r = (L \supseteq K \subseteq R, TG, NAC, PAC)$ consists of three model graphs L, K and R typed over TG. They are called left-hand side (L), intersection (K), and right-hand side (R). In addition, there are NAC and PAC, two sets of negative and positive application conditions. They are used to restrict rule applications by forbidding or requiring context patterns. Examples for rules are given in Figure 3.

A rule r can have several matches ("occurrences") in a model M. A match is a copy of L in M. Actual rule arguments form a partial match that has to be completed. Rule nodes may have more general types than corresponding graph nodes. A rule r is *applicable* at match m if m fulfills the *dangling condition*: If model nodes are deleted by a rule, all their incident edges have to be in the match as well. Moreover, the match can be extended by each positive application condition in PAC but not by any negative one in NAC. The *effects* of applying a rule r using match m in M can be described as follows: All elements in $m(L \setminus K)$ are deleted and a new copy of $R \setminus K$ is added. In addition, attribute values may be changed by instantiating attribute expressions of the right-hand side R and evaluating them.

Several rules can be composed to one rule such that their actions are performed concurrently. Therefore, the composed rule is called *concurrent* rule. Roughly speaking, a concurrent rule combines all actions of the original rules. Sequences of two actions that create and subsequently delete the same element, however, are factored out. Application conditions of subsequent rules are shifted to the beginning. If an application condition cannot be checked at the beginning (since an element is missing), it does not occur in the concurrent rule. Details of the construction of concurrent rules can be found in [10].

4 Generation of Edit Rules

In this section, we describe how to derive a set $\mathcal{R}$ of elementary edit rules from a given meta-model which we assume to be the effective meta-model w.r.t. a particular model editor. We define four kinds of edit rules for the *creation*, *deletion*, *moving* and *changing* of model elements. In the following, we mainly focus on the generation of creation rules since their generation process is most complex. The main design decision of our approach is that all generated edit rules are *consistency-preserving* w.r.t. the effective meta-model, i.e., if applied to consistent models, the resulting models are consistent as well. A consistency-preserving node creation rule usually comprises a number of primitive operations which, altogether, create a minimal graph pattern leading again to a consistent model.

In the following, we describe how creation rules are generated for a given meta-model. We begin with the generation of basic node creation rules. Subsequently, we show how these rules are to be supplemented such that mandatory children (see Section 3.1) are also created and all created nodes are connected to their mandatory neighbors in a single step.

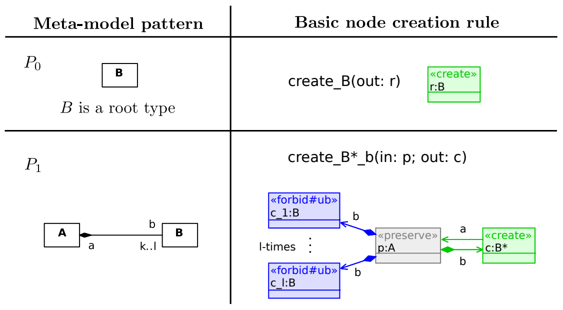

Figure 4. Generation of basic node creation rules

Creation rules. For each non-abstract root type B, a *node creation rule* is generated. This rule creates a single node of type B (see meta-model pattern P_0 in Figure 4).

For each node type B with an incoming containment edge type b, a rule according to pattern P_1 in Figure 4 is generated. This rule creates a node of type B - if non-abstract - and connects it immediately to its container. The notation $B*$ means that we derive such a rule for each concrete subtype of B as well.

If containment edge type b has a bounded multiplicity with upper bound l a NAC with l outgoing edges of type b is generated; it checks whether the parent node p has already the maximum number of outgoing edges of type b. If b has an associated opposite edge type a, edges of types b and a are created in pairs. Note that all figures show only the largest pattern/rule variants.

Basic node creation rules have to be extended by mandatory children since a node can recursively have (indirect) mandatory children and since our intention is to create all mandatory children by a single rule application. The supplementation is performed by subroutine SUPPLEMENTMCCREATION(Rule r, Node n), s. Figure 5. Each creation rule r for a node of type B is supplemented for each (inherited) outgoing containment edge type c of B with a multiplicity property *required* referencing a concrete node type C (see meta-model pattern P_2). Rule r is then further extended such that all mandatory children mc_1, ..., mc_k of n are created as well. Additionally, created nodes mc_1, ..., mc_k are immediately connected to their parent n via the respective containment edges of type c. Opposite edges are created if necessary. This subroutine has to be recursively executed to cover all (indirectly) connected mandatory children.

Rule create_StateMachine in Figure 3 is an example of an mc-supplemented rule. Initially, a node of type *StateMachine* is created. It has to be supplemented with a node of type *Region* and a containment edge of type *region* since this type is *required*.

In order to preserve multiplicity invariants defined by the effective meta-model, each created node must be immediately connected to its *mandatory neighbors*. We refer to extended rules which create these connections as *mn-*

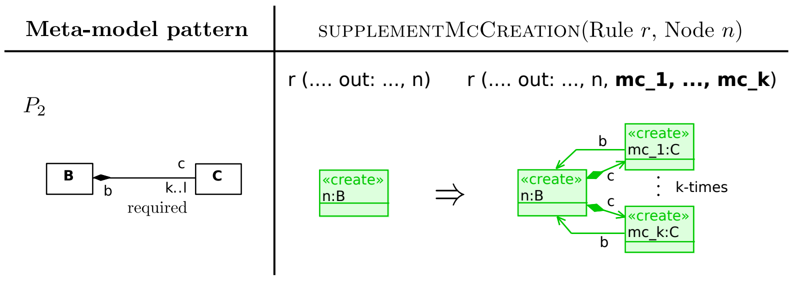

Figure 5. Supplementing the creation of mandatory children

supplemented node creation rules. This supplementation is performed by subroutine SUPPLEMENTMNCONNECTION(Rule r, Node n, EdgeType c), see Figure 6. If edge type c has an opposite edge type b, opposite edges are created in pairs. Moreover, a NAC is created for each mandatory neighbor mn_i (with $i \in \{1, ..., k\}$) prohibiting a connection of mn_i to m nodes of type B via edges of type b. Furthermore, values of (inherited) attributes of created nodes are set within a node creation rule since we conceptually treat them like mandatory neighbors. This supplementation has to be applied for all nodes created in a node creation rule. An example for this kind of supplementation is rule create_Transition_transition which does not only create a new transition, but also edges of type *source* and *target* to its mandatory neighbors as well as their opposites.

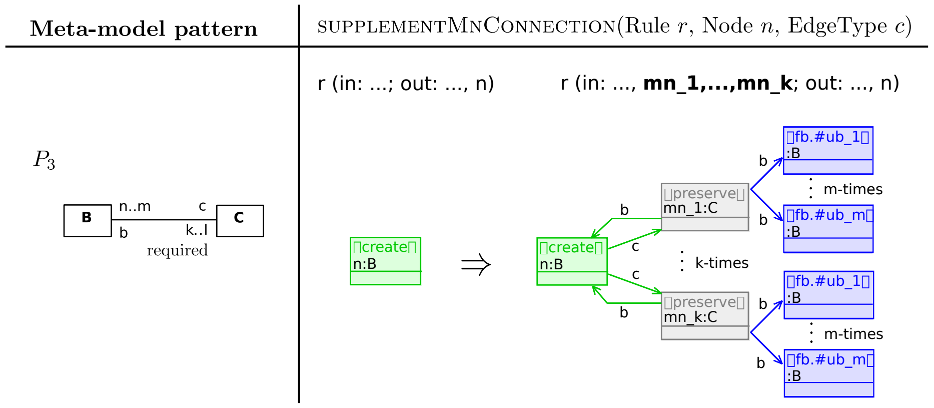

Figure 6. Supplementing the connection of mandatory neighbors

Since all generated rules are assumed to be applied injectively, there may be models that cannot be created with the generated rules so far. Missing rules can be generated by merging nodes of the same type if multiplicities do allow this variant. This merge construction is done after supplementation. Each merge variation leads to a further node creation rule. A simple example is shown by rule create_Transition_transition_1 in Figure 3, a variant of rule create_Transition_transition. A transition is created whose edges of types *source* and *target* lead to the same *State* node, i.e. this rule creates a "loop" pattern.

Moreover, it can happen that a *required* containment edge type points to a target node type with subtypes. Such a type graph cannot be flattened without using additional well-formedness rules. This requires a concrete rule variant for each possible combination of concrete types. In Figure 7, we need at least k containment edges of type b. Their targets, however, can have types B, C and D. The rule variants have to cover all possibilities.

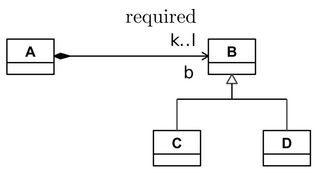

Figure 7. A critical multiplicity

To cover occasionally occurring meta-model patterns like cycles or parallel paths (i.e. two paths having the same source and target node) consisting of a mixture of *required* non-containment edges and *non-required* containment edges (see 3.2), we need a final post-processing step. For each identified cycle or parallel path, we identify the set of creation rules that cover it. These rules are brought into a suitable order according to causal dependencies and are composed to a concurrent rule. An example can be found in [1].

If a non-containment edge type b does not have a fixed multiplicity, then an *edge creation rule* is derived. Such a rule takes two parameters as input, namely the source and target nodes `s` and `t` of the new edge. If necessary, an opposite edge is also created. Additional NACs ensure that upper bounds have not already been reached.

Further kinds of edit rules. For each creation rule an inverse rule is generated, performing *deletion*. To invert a rule, its left and right-hand sides are exchanged. NACs which prohibit exceeding upper bounds are not needed. Instead, PACs are generated to ensure lower bounds, i.e., nodes and edges may be deleted as long as lower bounds are met. An example node deletion rule is shown in Figure 8. It deletes a *Region* from a *StateMachine*. In order to not violate the lower bound of edge type *region* (which has a multiplicity of [1..*], see Figure 2), the selected *Region* can only be deleted if the *StateMachine* contains at least one other *Region*.

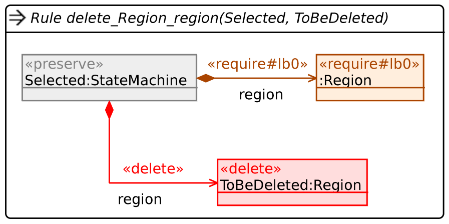

Figure 8. A sample deletion rule

Move and change rules re-structure the relations between existing model nodes. While a *move rule* moves an instance node from container to another one, a *change rule* just changes a link of a node. Lower and upper bound checks are inserted to ensure *no-lower-bound-violation* of the old reference links and *no-upper-bound-violation* of the new reference links.

Limitations of the approach. In general, there are combinations of multiplicities which cannot be instantiated (examples are shown in [15,29]). For meta-models that cannot be instantiated because of certain *required*-cycles, our generation algorithm does not terminate. Since we want to have a clear and efficient (in particular terminating) generation approach, we require an easy to check criterion which is not too limited to cover effective meta-models occurring in practice: *We do not allow meta-models having edge cycles with multiplicity pattern* required*, irrespectively of edge directions.* Such cycles do not allow a clear order of element creation and would lead to large creation rules, if any. Those rules would hardly specify edit operations. Small cycles of size ≤ 2, however, are supported (as already described above, see rule create_Transition_transition_1 in Figure 3). They are meaningful in effective meta-models. In the following, we restrict our considerations to meta-models obeying the restriction above, i.e., we also assume a corresponding restriction of type graphs.

Consistency-preservation and completeness of generated rules. Given a type graph T *with restricted multiplicities*, a rule is *consistency-preserving* if it transforms each consistent model graph to which it is applicable into a consistent model graph again. Our generator produces consistency-preserving rules only. An argumentation for this result can be found in Section 7.3.4 in [18].

A modeling language is defined by a set of models. Let $L(T)$ be the language consisting of all models that are consistent w.r.t. T. A set $\mathcal{R}_{Cre}$ of creation rules is *complete* w.r.t. T if every consistent model $M \in L(T)$ can be constructed from the empty model $\emptyset$ by exclusively using rules available in $\mathcal{R}_{Cre}$. Vice versa, a set $\mathcal{R}_{Del}$ of deletion rules is *complete* w.r.t. T if every consistent model $M \in L(T)$ can be reduced to the empty model $\emptyset$ by exclusively using rules available in $\mathcal{R}_{Del}$. Our generator produces a complete set of creation rules since every model graph of $L(T)$ can be partitioned into smaller graph fragments such that there is a sequence of rule applications creating the graph structure fragment-by-fragment. A detailed argumentation can be found in Section 7.4 in [18].

5 Evaluation

Our objective is to support tool developers at specifying consistency-preserving edit operations to be used to adapt MDE tools to domain-specific needs. This task should be highly automated. Moreover, the obtained edit rules should specify operations for conveniently editing domain-specific visual models. Consequently, we have evaluated our approach w.r.t. the following two research questions: **Q1:** *How limiting are our meta-model restrictions?* **Q2:** *Are the generated edit rules meaningful from the developer's point of view?*

Case studies. We studied four modeling languages for which (1) a perfect meta-model and (2) a visual editor are available. Table 1 presents an overview of the selected case studies. Ecore models can be considered as design-level class diagrams. They are widely used for various purposes in the Eclipse Modeling Project [8], a visual editor is available within the Ecore Tools [9]. The Simple Web Modeling Language (SWML) [5] is a domain-specific language which aims at defining platform-independent models for a specific kind of web applications. Feature models are typically used to define variability in software product line engineering. They have an intuitive tree-like syntax which is supported by the widely used feature modeling environment FeatureIDE [30]. A meta-model is presented in [7]. Concerning UML state machines, we selected the subset of the UML Superstructure Specification [24] which is shown in Figure 15.2 and analyzed how elements of these types are edited in MagicDraw [23]. Details of the case studies can be found in [1].

			Standard MM.			**Effective MM.**		
	Modeling Lang.	**Visual Editor**	#nt.	#et.	#wf.	#nt.	#et.	#wf.
I	Ecore	EcoreTools 3.0.1	19	26	37	16	15	5
II	SWML	Gen. with GMF 1.6.0	11	10	-	11	10	-
III	Feature models	FeatureIDE 2.6.1	8	12	3	6	8	2
IV	UML state mach.	MagicDraw 18.1	14	21	17	14	19	11

Table 1. Overview of the selected case studies

Evaluation setup. For each case study, we constructed the effective meta-model by reducing the perfect meta-model according to the effective level of consistency implemented by the respective visual editor. A typical reduction is the relaxation of multiplicities. E.g., the feature-related meta-model presented in [7] states that a feature group comprises at least two features by a multiplicity [2..*]. Although this is reasonable from a conceptual point of view, FeatureIDE offers the capability to create a group with a singleton feature only. Thus, the respective lower bound has been relaxed to [1..*]. Most notably, however, most of the additional well-formedness rules (*#wf.*) are neglected in effective meta-models (see Table 1). Many rules address the well-formedness of String expressions such as Boolean formulas over feature variables. Moreover, editors often do not support all language constructs defined by a DSML. In such a case, the effective meta-model is incomplete w.r.t. to the perfect meta-model in the sense that some node types (*#nt.*) and edge types (*#et.*) are not included (see Table 1). FeatureIDE, for example, does not support the visual modeling of cross-tree constraints as intended by its perfect meta-model. Using effective meta-models as input, our generator produces edit rules implemented in Henshin [3].

Limitations of the approach (Q1). For Q1, we are interested in whether effective meta-models contain consistency constraints which are not supported by our approach. If so, we are further interested in the manual effort which is required to manually adapt a generated rule set. As shown by Table 2 in column *#Unsp.Mult.*, none of the studied effective meta-models contains unsupported combinations of multiplicities, i.e. *required-cycles*, which are not supported by our generation algorithm, never occur. The number of generated edit rules is listed in column *#Gen.*, column *#Man.* lists the number of rules which have to be adjusted after the generation. The reason for manual adaptations of the generated rules is that well-formedness rules expressed in OCL are not yet supported by our algorithm. A few of them are still present in effective meta-models (see the last column of Table 1). An overview of the amount of manually adapted rules, on average 13%, is presented in Table 2. Typically, a few of the generated edit rules have to be complemented by additional application conditions. In FeatureIDE, e.g., features and feature groups must be organized in a strictly hierarchical way, violations of this well-formedness constraint have to be prevented.

	Q1			Q2			
	#Unsp. Mult.	#Gen.	#Man.	$\#(\mathcal{R} \cap \mathcal{E})$	$\#(\mathcal{R} \cap_{\approx} \mathcal{E})$	$\#(\mathcal{R} \setminus \mathcal{E})$	$\#(\mathcal{E} \setminus \mathcal{R})$
I	-	67	6	64	1	4	8
II	-	38	-	30	6	-	2
III	-	16	3	8	3	8	5
IV	-	66	18	57	18	2	27

Table 2. Overview of the evaluation results

Suitability of the obtained edit rules (Q2). Concerning Q2, we compare the set $\mathcal{R}$ of elementary edit rules finally obtained by our approach with the set $\mathcal{E}$ of rules specifying edit commands which are offered by the respective editor. We assume that these are meaningful from a modeler's point of view. Note that we specified the rules in $\mathcal{E}$ by hand. Table 2 summarizes the results. Columns $\mathcal{R} \cap \mathcal{E}$ and $\mathcal{R} \cap_{\approx} \mathcal{E}$ show the amount of identical and similar edit operations. Columns $\mathcal{R} \setminus \mathcal{E}$ and $\mathcal{E} \setminus \mathcal{R}$ summarize the amount of edit rules which are exclusively available in $\mathcal{R}$ and $\mathcal{E}$, respectively.

Most edit rules in $\mathcal{E}$ are specified by edit rules available in $\mathcal{R}$. Some of them, usually deletion rules, are not completely identical but lead to slightly different effects. A few deletion operations are rather complex in the sense that they delete larger model fragments consisting of an element and its mandatory children. For example, if an *EClass* is deleted in the Ecore diagram editor, *EAttributes* and *EOperations* contained by this *EClass* as well as outgoing and incoming *EReferences* to other *EClass*es are deleted as well. In contrast to that, our deletion rule assumes that an *EClass* can only be deleted if it is empty and has no inter-relations. A complex deletion rule, however, can be generated by inverting creation rules (see Section 4). Moreover, we found some operations in $\mathcal{E}$ which are not covered by $\mathcal{R}$ ($\mathcal{E} \setminus \mathcal{R}$). These rules can be considered as optional since their effect can also be achieved by applying a sequence of edit rules in $\mathcal{R}$. For example, FeatureIDE offers the possibility to create a new feature above a selected one. Using edit rules of $\mathcal{R}$, we create a new feature as a leaf node and then move the created feature to the designated position within the feature tree. Finally, there are some edit rules in $\mathcal{R}$ not having correspondents in $\mathcal{E}$ ($\mathcal{R} \setminus \mathcal{E}$). Typically, only a small subset of move operations is implemented in visual editors. The Ecore diagram editor, for instance, offers the possibility to move an *EAttribute* to another *EClass* while moving *EClasses* between *EPackages* is not supported.

Threats to validity. A threat to the external validity of our results is that the selected case studies may not be representative. However, we selected modeling languages which differ significantly from each other and cover a broad range of application domains. Moreover, we selected visual editors having substantially different origins; from the open source community (Ecore diagram editor), from academia (SWML and FeatureIDE) and a commercial product (MagicDraw). An internal threat to validity is our manual deduction of edit rules from existing editors. Likewise, the reduction of a perfect meta-model to become the effective is done manually, too.

6 Related Work

We consider other approaches for edit rule generation on the one hand and, w.r.t. creation rules, compare to further approaches for creating meta-model instances on the other hand.

The work closest to ours has been presented in the context of delta-oriented implementation of model-based software product lines (SPLs). Products of an SPL are generated by applying one or several deltas to the core version. A delta is basically a patch which consists of a sequence of edit commands. For a given DSML, a delta modeling language [13] must be engineered; it contains basically a set of edit operations (called "delta operations") for this DSML. To that end, Seidl et al. [27] present an approach and a supporting tool known as DeltaEcore to generate executable delta operations from EMOF-based meta-models, however, with different goals and assumptions compared to our work. In particular, they assume that the application of a delta will never fail and that SPL developers are responsible for specifying consistency-preserving deltas. In particular, they do not support any kind of multiplicities in meta-models.

Ehrig et al. [11] deduce graph grammar rules from meta-models. The generated set of rules is organized in three layers: Layer 1 rules create instances of meta-model classes, Layer 2 establishes mandatory relationships between elements. In this step additional elements are also created when necessary. Finally, Layer 3 rules establish optional relationships. Taentzer [29] extends the approach from restricted multiplicities to arbitrary ones. Using the concept of layered graph grammars obviously leads to inconsistent intermediate states since instance models are created in small steps. Hence, the generated rules do not implement consistency-preserving edit operations. Moreover, other kinds of edit rules are not generated in that approach at all.

Hoffmann and Minas [15] describe how to translate a class diagram into a so-called adaptive star grammar. Their generated rules use non-terminal symbols to direct the generation process. Small steps are performed leading to intermediate graphs with non-terminals. In the same vein, Fürst et al. [12] present an approach for generating meta-model instances using graph grammars with non-terminals.

Edit operations are indirectly addressed in some approaches which aim at generating instance models for a given meta-model. Virtually all of these approaches are based on the idea to systematically enumerate meta-model instances. Brottier et al. [6] describe an enumeration algorithm which is based on model fragments that must be specified manually. Other approaches use SAT-solvers such as the Alloy Analyzer [17] to systematically enumerate valid instances in a restricted search space. However, they do not identify which edit operations have to be applied to obtain instances.

7 Conclusion

In this paper, we present the main concepts for a rule generator which takes a meta-model with restricted multiplicities and yields a complete set of consistency-preserving edit rules. Their main purpose is to raise the abstraction level in model versioning. Concerning meta-models which are effectively used by model editors, our evaluation shows that the established meta-model restrictions are not severely limiting in practice. It also outlines possible directions for future work: The generator shall be extended to accept meta-models with well-formedness rules. Radke et al. [25] present how OCL constraints can be translated to application conditions, using nested graph constraints as intermediate representation. That work may be used to generate edit rules which also take well-formedness rules into account. The vision is the automated specification of a complete set of consistency-preserving edit operations for any effective meta-model which may be valuable not only for specific model versioning tasks but for model change management in general.

References

1. Accompanying material for this paper: (2015), `http://pi.informatik.uni-siegen.de/projects/SiLift/icmt2016/index.php`
2. Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: Why model versioning research is needed!? an experience report. In: Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS. vol. 9 (2009)
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: Model Driven Engineering Languages and Systems, pp. 121–135. Springer (2010)
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. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers (2012)
6. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Meta-model-based test generation for model transformations: an algorithm and a tool. In: 17th International Symposium on Software Reliability Engineering. pp. 85–94. IEEE (2006)
7. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex differences on feature models. Automated Software Engineering (2015)
8. Eclipse Modeling Project (EMP): (2015), `http://eclipse.org/modeling`
9. Ecore Tools - Graphical Modeling for Ecore: (2015), `http://www.eclipse.org/ecoretools`
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
11. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software & Systems Modeling 8(4), 479–500 (2009)
12. Fürst, L., Mernik, M., Mahnic, V.: Converting metamodels to graph grammars: doing without advanced graph grammar features. Software and System Modeling 14(3), 1297–1317 (2015)

13. Haber, A., Hölldobler, K., Kolassa, C., Look, M., Rumpe, B., Müller, K., Schaefer, I.: Engineering delta modeling languages. In: Proceedings of the 17th International Software Product Line Conference. pp. 22–31. ACM (2013)
14. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. 8(3), 231–274 (1987)
15. Hoffmann, B., Minas, M.: Generating instance graphs from class diagrams with adaptive star grammars. ECEASST 39 (2011)
16. IBM, Rational Software Architect: (2015), `http://www-03.ibm.com/software/products/en/ratisoftarch`
17. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
18. Kehrer, T.: Calculation and Propagation of Model Changes Based on User-level Edit Operations. Ph.D. thesis, University of Siegen (2015)
19. Kehrer, T., Kelter, U., Reuling, D.: Workspace updates of visual models. In: ACM/IEEE International Conference on Automated Software Engineering (ASE). pp. 827–830. ACM (2014)
20. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: 28th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE). pp. 191–201. IEEE (2013)
21. Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., Kappel, G.: A posteriori operation detection in evolving software models. Journal of Systems and Software 86(2), 551–566 (2013)
22. Mens, T.: On the use of graph transformations for model refactoring. In: Generative and transformational techniques in software engineering, pp. 219–257. Springer (2006)
23. No Magic, MagicDraw: (2015), `http://www.nomagic.com/products/magicdraw.html`
24. Object Management Group: Uml 2.4.1 superstructure specification. OMG Document Number: formal/2011-08-06 (2011)
25. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints focusing on set operations. In: 8th Intl. Conf. on Graph Transformation (ICGT). pp. 155–170. Springer (2015)
26. Rindt, M., Kehrer, T., Kelter, U.: Automatic generation of consistency-preserving edit operations for MDE tools. In: Demonstrations Track of the ACM/IEEE 17th Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS). CEUR Workshop Proceedings, vol. 1255 (2014)
27. Seidl, C., Schaefer, I., Aßmann, U.: DeltaEcore-A Model-Based Delta Language Generation Framework. In: Modellierung. pp. 81–96 (2014)
28. Selonen, P., Kettunen, M.: Metamodel-based inference of inter-model correspondence. In: 11th European Conf. on Software Maintenance and Reengineering (CSMR). pp. 71–80. IEEE (2007)
29. Taentzer, G.: Instance generation from type graphs with arbitrary multiplicities. Electronic Communications of the EASST 47 (2012)
30. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming 79, 70–85 (2014)