Generating Edit Operations for Profiled UML Models

Timo Kehrer, Michaela Rindt, Pit Pietsch, Udo Kelter

Software Engineering Group
University of Siegen
{kehrer,mrindt,pietsch,kelter}@informatik.uni-siegen.de

Abstract. Since many tools for model-driven engineering (MDE) can only process consistent models, the consistency must be guaranteed by all tools which modify models, such as model editors, transformers, merge tools, etc. This is a recurring challenge for tool developers.
An obvious solution of this challenge is to use a common library of consistency-preserving edit operations for modifying models. Typical meta-models lead to several 1000 edit operations, i.e. it is hardly possible to manually specify and implement so many edit operations. This problem is aggravated by UML profiles: Stereotyped model elements are implemented as complex data structures. This paper discusses several approaches to implementing edit operations on profiled models. Furthermore, it canvasses how to generate complete sets of specifications and implementations of edit operations.

1 Introduction

In model-driven engineering (MDE) models are the primary development artifacts. Models are modified by many different tools, including model editors, transformers, patch and merge tools [7], test data generators [11] etc. Many tools can only process consistent models; tool constructors are thus faced with the challenge that all tools must preserve the consistency of the models.

A general solution to the challenge described above is to use a library of edit operations, but many of the existing libraries do not guarantee the consistency of the models. In this paper we present a novel approach to specify and implement a library of consistency-preserving edit operations (Section 2).

Consistency-preserving edit operations (CPEOs) depend on the meta-model, i.e. they are not generic. The set of CPEOs available for a given meta-model must be *complete* in the following sense: It must be possible to construct any consistent model and to edit each consistent model to become another consistent model. Complete sets of CPEOs are quite large; their creation should therefore be supported by (meta-) tools. Section 2 introduces SERGE, our own tool to generate CPEOs, and discusses the limitations of automatically generating sets of CPEOs.

The challenge of defining CPEOs is aggravated in case of domain-specific modeling languages (DSMLs) which are defined using the UML profile mechanism, e.g. SysML [9] and MARTE [10]: Stereotyped model elements are implemented as complex data structures which create new consistency requirements (See Section 4).

The main contribution of this paper is an analysis of how profiles affect (sets of) CPEOs of the underlying non-extended meta-model. This is discussed in Section 5.

Several approaches are available for implementing CPEOs for profiled meta-models and for semi-automatically generating complete sets of CPEOs. Section 6 presents these approaches and discusses their respective advantages and disadvantages.

Section 7 summarizes the contributions of this paper.

2 Consistency-preserving Edit Operations

Meta-Models. A model is conceptually considered as a typed, attributed graph which is known as the abstract syntax graph (ASG). Meta-models, e.g. the UML meta-model, define the types of nodes and edges allowed in an ASG. Following the common practice in MDE, we assume that there is an object-oriented implementation of meta-models. Thus, nodes and edges of an ASG are represented by (runtime) objects and references between them.

Consistency of Models. An ASG is considered consistent if it complies to the definitions and constraints specified by its meta-model, notably constraints concerning hierarchies, relationships and multiplicities.

Standards such as the UML typically define strict, “ideal” consistency constraints; ASGs complying with them represent models which can be translated to source code and other platform-specific documents. Many model editors (e.g. the Eclipse UML Model Editor) use de facto less strict meta-models. For example, they enforce only multiplicity constraints which are required to produce a graphical representation of a model.

Edit Operations vs. Basic Graph Operations. Meta-models are just data models of models, they do not directly specify editing behavior. One obvious approach to modify models is to use ’basic graph operations’ on the ASG including *deleting*, *creating*, *moving* and *changing* elements, attributes or references.

Executing a single basic graph operation on an ASG can lead to a new state which violates the syntactic consistency. For example, a basic operation which creates a single UML *StateMachine* object leads to an inconsistent ASG: A *StateMachine* must always contain at least one *Region*. In contrast to this, a CPEO will create a *StateMachine* and create a contained *Region* at the same time. Thus, the model is transformed from one consistent state into another. This CPEO is *minimal* in the sense that it cannot be splitted into smaller parts which preserve the consistency of the model.

In general, an edit operation has an interface specifying input and output parameters. For example, the edit operation createStateMachine must be supplied with a container element in which the *StateMachine* is to be created, along with local attribute values for the new *StateMachine*. Objects created by an edit operation are handled as output arguments. Additionally, pre- and postconditions may precisely specify application conditions of an edit operation.

Basically, minimal CPEOs can be categorized into the same kinds of operations as basic graph operations, namely *creation*, *deletion*, *move* and *change* operations. However, minimal CPEOs usually comprise a set of basic ASG operations. An example for such an 'atomic compositions' of ASG elements is the creation of a *StateMachine* with at least one nested *Region* in a single transaction.

3 Generating Executable Specifications of Edit Operations

Comprehensive languages, such as the UML require large sets of edit operations. The manual specification of such sets of CPEOs for a given modeling language is very tedious and prone to errors.

This problem is addressed by our *SiDiff Edit Rules Generator (SERGe)* [12]. SERGe derives sets of minimal CPEOs from a given meta-model with multiplicity constraints. These sets are complete in the sense that all kinds of edit operations, i.e. create, delete, move and change operations, are contained for every model element, reference and attribute. Respectively, edit operations generated by SERGe cover multiplicity constraints and maintain the consistency of a model.

According to our knowledge, there are no other approaches for constructing CPEOs. There are approaches to create certain kinds of edit operations or grammars which can construct or modify models [1, 3, 6, 13]. However, they either do not support all types of modifications (e.g. *setting attribute values* or *moving elements*) or they lead to consistency violations. In sum, they were unusable for our own practical work.

SERGe uses the Eclipse Modeling Framework (EMF) [4]. Generated edit operations are realized as in-place transformation rules in the model transformation language Henshin [2, 5]. A Henshin transformation rule can specify model patterns to be found and preserved, to be deleted, to be created and to be forbidden. We will refer to these implementations of edit operations as *edit rules*.

Figure 1 shows the edit rule createStateMachine. This edit rule is generated by SERGe when processing the implementation of the UML2 meta-model in EMF Ecore. For the sake of readability, the rule shown in Figure 1 is a simplified version of the rule.

This example illustrates that a Henshin rule can define variables serving as input or output parameters. The input parameter *Selected* determines the context object to which an edit rule shall be applied. In our example, the context object will be the *Model* in which the *StateMachine* shall be created. The

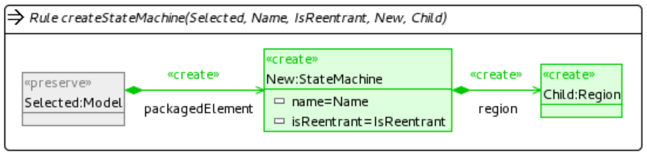

Fig. 1. Edit rule createStateMachine generated by SERGe

StateMachine object and its mandatory *Region* will be returned as output parameters *New* and *Child* when the rule is applied. Additionally, the *name* and the property *isReentrant* of the *StateMachine* to be created must be provided by the input value parameters *Name* and *IsReentrant*, respectively.

Manual Adaptions of Generated Operations. SERGe is not capable of interpreting arbitrary well-formedness constraints (OCL **??** Constraints) attached to a meta-model element. Hence, some of the generated edit rules have to be complemented by additional application conditions.

For example, the UML meta-model specifies that all the members of a *Namespace* (which includes the packaged elements of a *Model*) are distinguishable by their names. Thus, a *StateMachine* can only be created in a *Model* if there is no *NamedElement* with the same name. This precondition can be implemented in Henshin by a negative application condition (NAC) as shown in Figure 2.

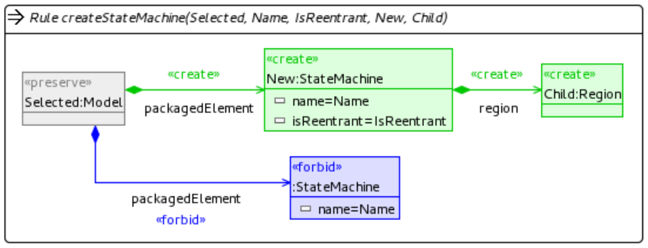

Fig. 2. Manually adapted edit rule createStateMachine

Conclusion. We can conclude that the most important innovation of SERGe is that the generated edit operations comply to multiplicity constraints and preserve the consistency of a model in this respect. If required, the generated edit operations can be extended manually additional well-formedness constraints; this usually requires only a limited effort.

However, SERGe currently does not address the generation of edit operations for UML profiles. Several design variants to extend a generator such as SERGe to profile definitions will be discussed in the remainder of this paper.

4 The UML Profile Mechanism

UML profiles provide a light-weight approach to implement domain-specific languages by reusing existing meta-models.

Profile Definition. Profiles are basically defined as follows: A profile must import the base meta-model which contains the reused element types. Principally, any MOF-based meta-model can be extended by a profile definition. In practice, the UML meta-model serves as the base meta-model. Figure 3 shows an excerpt of the SysML [9] profile definition. The profile defines stereotypes which extend one or more of the imported element types. These extended classes are called 'meta-classes'. The attribute *required* of a meta-class extension defines whether an instance of the extending stereotype must be attached to any instance of this meta-class. A stereotype can have new attributes or references, the so called 'tagged values'.

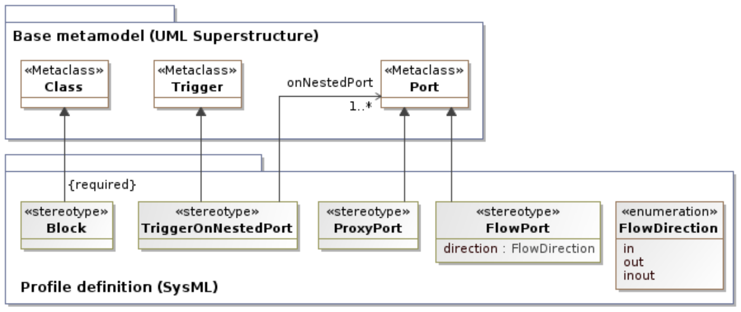

Fig. 3. SysML-Profile excerpt

Profile Application. A profile can be applied to a model which is an instance of the base meta-model, and later be revoked. The application or revocation of a profile can also be regarded as an edit operation: It causes stereotypes to be added to or to be removed from appropriate model elements of a model.

The UML profile mechanism is designed in a way that all data related to a profile are separated from the extended model; i.e. one or more profiles can be applied to a UML model without destroying its previous structure. Thus, the extended UML model always remains processable by UML tools.

An example of a profile application is shown in Figure 4; a very simple SysML model is shown in concrete syntax (left) and abstract syntax (right). It illustrates how stereotype objects of type *Block* and *FlowPort* (colored in light gray) are attached to instances of the meta-classes *Class* and *Port*, respectively.

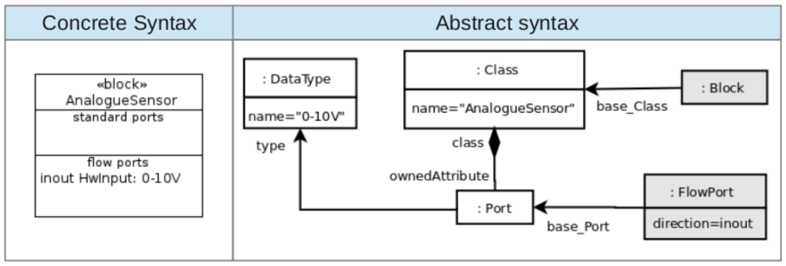

Fig. 4. Sample SysML model in concrete syntax (left) and abstract syntax (right)

5 Edit Operations for Profiled UML Models

In this section we analyze how UML profile definitions influence the set of CPEOs. The complete set of edit operations can be divided into four disjoint subsets which are described below.

A. Edit Operations modifying UML Model Elements. The first set of edit operations operates on model elements that are instances of meta-classes of the base meta-model, i.e. the UML meta-model. For the sake of readability, we refer to them as (pure) **UML model element operations**. In principle, no knowledge about applicable profiles is required to generate these operations.

Nevertheless, these model elements can have stereotypes. Although the effect of a UML model element operation does not depend on whether or not there is a stereotype, it can be necessary to omit or rename some of the generated edit operations, notably in case of *required* stereotypes. Thus, we further divide the set of UML model element operations into two main subsets:

1. Edit operations which *create* or *delete* UML model elements and their containment references.
   These edit operations must be omitted if the type of the involved model element has a *required* stereotype in the profile definition. In this case, the base model element and the stereotype object have to be handled consistently by hybrid edit operations (s. Section 5.D). Other edit operations which modify the containment hierarchy, notably relocations of model elements caused by *move* operations, are not omitted.
2. Edit operations which *change* attributes or non-containment references of model elements. They are independent on whether or not a profile is applied. It can be helpful to rename these operations for the sake of understandability, e.g. from setClassIsAbstract to setBlockIsAbstract.

In both cases, edit operations are omitted if they operate on instances of meta-model elements which are excluded by the profile definition. For example, SysML does not support interaction diagrams as defined in UML2. Thus, edit operations modifying elements of type *Interaction*, *InteractionFragment* or *Lifeline* are omitted.

B. Edit Operations modifying Tagged Values. This subset comprises edit operations which *change* attributes or references of stereotypes. For the sake of simplicity, we refer to this subset as **tagged value modifications**. These edit operations modify only parts of a model, specifically the stereotype instances of the profile. They can have arguments whose type is defined by the base meta-model; these arguments remain unchanged.

Tagged value modifications are easy to define and implement if their domain is a primitive data type. For example, the edit operation setBlockIsEncapsulated simply sets the tagged value *isEncapsulated* of a SysML *Block*. Tagged value modifications are more complicated if they change references of stereotype objects. Here, multiplicity constraints which define mandatory neighbours or children must be taken into account. These cases can be handled in the same way as modifications of references on UML model elements (see Section 2).

C. Edit Operations for Stereotype Application. Stereotypes are applied to or removed from an UML model element by **Stereotype Applications**. From a users' point of view, this can appear as a conversion of a model element to another type. On the ASG level, a stereotype application is an edit operation that *creates* or *deletes* a stereotype object together with the reference to an instance of its base meta-class.

These edit operations are not permitted for stereotypes that are declared as required for their base meta-class. In such cases they are omitted from this set of edit operations. For example, the stereotype application operations applyStereotypeBlock and unapplyStereotypeBlock are not applicable to SysML models. The stereotype *Block* must always be created or deleted together with an instance of its base meta-class *Class*, but not without. This kind of modification is achieved by a hybrid edit operation.

D. Hybrid Edit Operations for Required Stereotypes. Hybrid edit operations concurrently modify instances of base meta-classes *and* stereotype instances. Typically, they *create* or *delete* model elements that have required stereotypes. An example is given in Section 5.C; creating a SysML *Block* requires the creation of a UML *Class* to which it must be attached as stereotype.

6 Generating Edit Operations

The previous section introduced 4 different kinds of edit operations. The complete set of operations which is necessary to consistently edit profiled UML models, is the union of these 4 sets and represented by the dotted rectangle in Figure 5.

The edit operations specified in subsections 5.A can be generated by SERGe. The same is true for edit operations specified in subsections 5.B and 5.C because profile definitions are handled as usual MOF-based meta-models.

However, some extensions are necessary for the generation of hybrid edit operations (Section 5.D). Generally, hybrid edit operations can be implemented using two different approaches: A *higher-order transformation approach* and a *meta-model driven approach*.

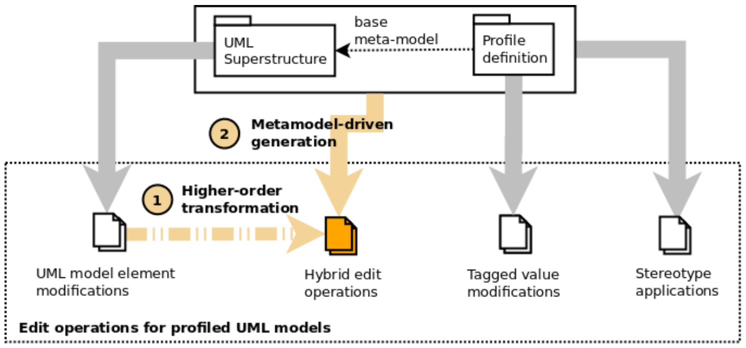
UML Superstructure
base meta-model
Profile definition
2 Metamodel-driven generation
1 Higher-order transformation
UML model element modifications
Hybrid edit operations
Tagged value modifications
Stereotype applications
Edit operations for profiled UML models

Fig. 5. Generation of different types of edit operations

A. Higher-order Transformation Approach. This approach uses a set of existing UML edit operations as primary input (see arrow 1 in Figure 5). It does not matter whether the edit operations have been generated and/or constructed manually. The basic idea is to modify them by adding appropriate stereotypes.

We assume that edit operations are defined as executable specifications in the form of Henshin transformation rules as explained in Section 2. Henshin rules are technically represented as models and can be transformed automatically, i.e. their modification can be considered as a higher-order transformation (HOT).

We have implemented this HOT in Henshin as follows: Basically, stereotypes as defined by the profile are applied to all instances of a UML base meta-class which occur in the given set of UML edit rules. A parametrized HOT rule is provided in order to attach an instance of a stereotype. The necessary parameters are the stereotype and the UML meta-class. We have implemented three variants of this HOT rule: They attach stereotype objects to UML model elements which are (1) to be created, (2) to be deleted and (3) to be preserved by an edit rule. A working example of a HOT rule can be found at [12].

Since a UML meta-class can be extended by several stereotypes, different variants for every stereotype will be created. Such variants can also consist of combinations of multiple stereotypes, if more than one meta-class is contained in the given UML edit rule. Thus, the scheduling algorithm which applies the HOT rules with appropriate invocation arguments is responsible for generating all possible combinations of stereotype attachments.

A big advantage of this approach is that efforts of manual adjustments on UML edit operations are not lost. A disadvantage is that it supports only simple profiles: It cannot handle stereotypes which have mandatory references to other stereotypes or UML model elements. However, important profile-based standards such as SysML and MARTE rarely contain such scenarios.

B. Metamodel-driven Approach. This approach does not require any previously generated sets of edit operations, all operations are generated from scratch; the profile and the base meta-model are the only input data here (see arrow 2 in Figure 5).

This approach can consider multiplicity constraints of (non-) containment references which emanate from stereotypes. However, in contrast to the HOT approach, all manual adjustments on an existing set of UML edit operations (modifications, creations and deletions) are lost. Manually created edit operations for the base language are not automatically adapted.

Two alternative patterns are available for implementing a hybrid operation; Figures 6 and 7 illustrate them using the creation of a SysML *Block* as an example:

1. **Sequentially boxed operations** (Figure 6): Here, the UML edit operation and the profile application operation are applied in sequential order; initially, a class instance is created. In a next step the required stereotype is added.
2. **Concurrent operation** (Figure 7): Basically, all edit operations used separately in the first approach are 'merged' into one. For Henshin transformation rules, such a merge can be implemented by concurrent rule construction.

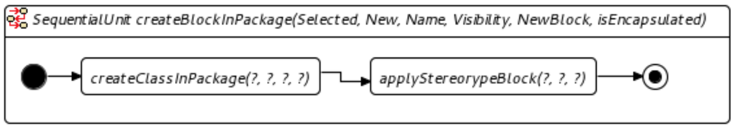
SequentialUnit createBlockInPackage(Selected, New, Name, Visibility, NewBlock, isEncapsulated)
createClassInPackage(?, ?, ?, ?)
applyStereorypeBlock(?, ?, ?)

Fig. 6. Implementing a hybrid edit operation using a sequential transformation unit

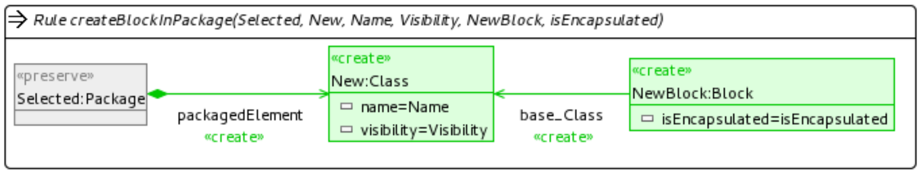

Fig. 7. Implementing a hybrid edit operation using a single transformation rule

7 Conclusion

This paper has presented our approach to construct complete sets of consistency-preserving edit operations on models. In contrast, existing approaches for generating edit operations do not support consistency preservation and DSMLs using UML Profiles such as SysML or MARTE.

We addressed an important practical requirement: How to consistently maintain operation sets for the base language without profiles and for the profiled language. Our solution is based on a clear separation of all edit operations in 4 categories. The most complex type of edit operations are hybrid ones; we showed that they can be implemented in such a way that manual optimizations of the base edit operations can be preserved.

Acknowledgement. This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

References

1. Alanen, M.; Porres, I.: A relation between context-free grammars and meta object facility metamodels; Technical Report 606, TUCS Turku Center for Computer Science; 2003;
2. Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. MoDELS 2010, Oslo; LNCS 6394, Springer; 2010
3. Ehrig, K.; Küster, J.M.; Taentzer, G.; Generating instance models from meta models; SoSym Volume 8:4, p.479-500; 2009
4. EMF: Eclipse Modeling Framework; http://www.eclipse.org/emf; 2012
5. EMF Henshin Project; http://www.eclipse.org/modeling/emft/henshin
6. Hoffmann, B.; Minas, M.: Generating instance graphs from class diagrams with adaptive star grammars. Intl. Workshop on Graph Computation Models, 2011
7. Kehrer, T.; Kelter, U.; Taentzer, G.: Consistency-Preserving Edit Scripts in Model Versioning; in: Proc. 28th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE 2013); ACM; 2013
8. Object Constraint Language: Version 2.0; OMG, Doc. formal/2006-05-01; 2006
9. Systems Modeling Language: Version 1.3; OMG, Doc. formal/2012-06-01; 2012
10. UML Profile For Marte - Modeling And Analysis Of Real-time Embedded System: Version 1.1; OMG, Doc. formal/2011-06-02; 2011
11. Pietsch, P.; Shariat Yazdi, H.; Kelter, U.: Generating Realistic Test Models for Model Processing Tools; p.620-623 in: Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11); ACM; 2011
12. The SiDiff EditRule Generator - A tool to automatically derive consistency-preserving edit operations of any ecore meta model; http://pi.informatik.uni-siegen.de/Mitarbeiter/mrindt/SERGe.php; 2012
13. Taentzer, G.: Instance Generation from Type Graphs with Arbitrary Multiplicities; in: Electronic Communication of the EASST 47; 2012