Consistency-Preserving Edit Scripts in Model Versioning

Technical Report
2013-11-12

Timo Kehrer, Udo Kelter
Department Elektrotechnik und Informatik
Universität Siegen, Germany

Gabriele Taentzer
Fachbereich Mathematik und Informatik
Philipps-Universität Marburg, Germany
1 Introduction

Collaborative work and version management must be supported by sophisticated tools. These tools must be adapted to the types of documents used in a development methodology. In model-based software development, models are primary artifacts which are frequently changed.

Basic services of tools supporting version management include comparison, merging, and patching of models. Conceptually, these services are based on fundamental notions including difference (or delta), conflict etc. Comprehensive conceptual frameworks for versioning have been developed in the context of source code versioning [4, 28, 40]. These frameworks define a directed delta (or difference) as a sequence of (elementary) edit steps \( s_1 \ldots s_m \) which, when applied to a document version \( v_1 \), yields another version \( v_2 \). An edit step invokes an edit operation and supplies appropriate actual parameters; all edit operations used in a directed delta must be applicable to the type of the document being modified.

These definitions leave open how documents are represented conceptually and which edit operations are available for modifying a document. A classical approach is to use textual representations of documents and text editing operations. It has often been suggested to conceptually represent source code as abstract syntax trees (AST) and to use (elementary) tree editing operations in edit steps. More advanced approaches propose to use non-elementary transformations such as refactorings as edit operations for ASTs. Directed deltas based on complex edit operations enable “structural merging” [7, 26, 28] which enables a better conflict detection compared with textual merging. While the advantages of structural merging are undisputed, appropriate tool support is difficult to implement and hard to find in practice.

The above considerations also apply to models. It is state of the art to consider models conceptually as abstract syntax graphs (ASG) and to use generic graph operations as edit operations for ASGs. As an example, Figure 1 shows two revisions of a simple UML class model. The base version A has been edited to become the revised version B as follows:

1. A generalization relationship has been created such that class Developer specializes Person.
2. Attribute name has been pulled up along this generalization [11].

If we compare the ASGs before and after the above edit steps, we will observe a large number of basic graph modifications, which are indicated as colored nodes and edges in Figure 1. Such a low-level difference is reported by model comparison tools currently available. These tools cover only the first two processing steps of the model differencing tool chain shown in Figure 2. Initially, a matching algorithm identifies corresponding nodes and edges in both ASGs. Nodes and edges not involved in a correspondence are considered to be deleted or created (this is considered in more detail in Section 4). This simple example shows already that model differences can be hard to understand for modelers which do not know the meta-model of a modeling language.

Generic graph operations lead to further problems because they can violate consistency constraints in ASGs. For example, creating a Generalization object without connecting it to Class objects by references general and specific leads to an inconsistent model which cannot be visualized textually or graphically and which cannot be processed by code
Figure 1: Original model A and its revision B: Abstract and concrete syntax
generators. This causes serious problems in 3-way-merging or patching scenarios: conflict resolution in 3-way-merging requires some edit steps involved in conflicts to be excluded from the final merge result [26], edit steps in patches cannot be executed if a referenced model element can not be found in the target model [20, 23]. All other edit steps which depend on a deleted one must be identified and removed automatically. Ultimately, only a subset of the edit steps contained in a directed delta is actually executed in these cases. Therefore, directed deltas for models should use only consistency-preserving edit operations which transform a model from one consistent, displayable state to another.

One contribution of this paper is the definition of an extended kind of directed delta which we call edit script. While a directed delta is basically a sequence of edit steps, an edit script is a semi-ordered set of edit steps and comprises all information about arguments and dependencies of edit steps as well as interfaces of edit operations (see Section 2). Another contribution of this paper is a comprehensive tool support which, given a set of edit operations defined as model transformation rules (see Section 3), creates a generator for consistency-preserving edit scripts; Starting from a given low-level difference (see Section 4), the generator finds applications of the given transformation rules in the low-level difference (see Section 5) and constructs an edit script (see Section 6). These extensions to state-of-the-art model differencing tools are colored in light gray in Figure 2.

Our approach assumes a complete set of consistency-preserving edit rules; Section 7 shows how we generate such a set from a meta-model. We have evaluated our approach in an empirical case study (see Section 8). Related work is discussed in Section 9. Section 10 concludes this paper.
2 Consistency-Preserving Editing of Models

Since our goal is to generate consistency-preserving edit scripts, this section will address the question how to define consistency-preserving edit operations for models.

2.1 Meta-Models

Conceptually, the structure of a model is considered as a typed, attributed graph, which is known as abstract syntax graph (ASG) of this model. We assume the types of nodes and edges allowed in the ASG and their properties to be defined by a meta-model, e.g. the UML [38] or SysML [34] meta-model. In addition, a meta-model defines well-formedness rules which restrict the allowed values and/or instance structures in the ASG. These rules are usually formulated using the Object Constraint Language (OCL) [30].

ASGs can be implemented in various technical frameworks. In this paper, we assume models to be implemented using the Eclipse Modeling Framework (EMF) [9]. EMF is one widely used technology which supports constructing runtime representations of models in the Java language. Nodes and edges of the ASG of a model are represented by run-time objects and references between them. Of course, all concepts can be transferred to other technologies and frameworks.

2.2 Edit Operations

Meta-models do not directly specify “editing behavior”, i.e. edit operations which modify models. One obvious approach to modify models is to use elementary operations such as creating or deleting nodes and edges in the ASG, changing the values of attributes etc. These operations are “low-level” in the sense that they consider an ASG as an ordinary directed graph and operate on this graph without considering well-formedness rules. We refer to all these basic operations as change actions.

The execution of a single action can lead to a “model” violating well-formedness rules, i.e. a model that is inconsistent. In contrast to basic change actions, edit operations preserve the consistency of models to which they are applied. We assume that each edit operation has an interface specifying input and output parameters as well as an implementation (as shown in the bottom left part of Figure 4).
3 Implementing Edit Operations in Henshin

Edit operations can be implemented by in-place model transformations. The model transformation language Henshin [1, 15] is well-suited for this purpose. It is based on graph transformation concepts [10] which can be exploited to reason about conflicts and dependencies between edit operations.

We assume that edit operations are implemented using transformation rules in Henshin and refer to them as edit rules. As an example, the core part of a Henshin rule implementing the edit operation pullUpAttribute(p,g,n) is shown as edit rule in Figure 3. For the sake of readability, certain parts of the edit operation are omitted in this figure. In the complete rule, all attributes named n having the same type as p are also deleted from all other direct subclasses of cg. A complete specification of the edit operation pullUpAttribute (for a simplified UML meta-model) by Henshin rules can be found in [1].

Figure 3: Henshin rule implementing the edit operation pullUpAttribute(p,g,n)

The example shows that a Henshin transformation rule can define variables serving as input or output parameters. Here, p, g, and n are input parameters. Property p with name n is the attribute which shall be pulled up from class cs to superclass cg, but only if cg does not yet own an attribute with name n. As UML classes may have multiple superclasses, the concrete superclass cg of cs is determined by the generalization relationship g given as input parameter. p and g are object parameters, while n is a value parameter. Variables cs and cg are bound implicitly when pullUpAttribute is applied.

A Henshin transformation rule r is defined on the ASG and consists of the following components:

L \quad \text{The left-hand side is a graph pattern which specifies the pattern that has to be found in the model for applying } r. \text{ L may include checks of attribute values which are specified by constants or variables.}

R \quad \text{The right-hand side specifies a graph pattern which replaces } L \text{ if } r \text{ is applied. We assume a matching between } L \text{ and } R \text{ which matches all elements of } Cxt_r. \text{ Corresponding elements must be identical in } L \text{ and } R. \text{ They may include changes of attribute values specified by expressions including declared variables.}

Cxt_r \quad \text{is the intersection of } L \text{ and } R. \text{ It identifies that part which shall remain unchanged when } r \text{ is applied.}

r : L \rightarrow R \quad \text{is used as a shorthand notation for the above components. The arrow symbolizes a partial mapping from } L \text{ to } R.
4 DIFFERENCING OF MODELS

$PAC_r$, $NAC_r$ are sets of positive and negative application conditions $c$; each $c$ is an extension of $L$. In general, conditions $c$ are not graphs, but graph fragments. They also may contain attribute conditions over declared variables.

In addition, we use the following notations:

$Del_r$ is the fragment of $L$ which is deleted when $r$ is applied.

$Cre_r$ is the fragment of $R$ which is created when $r$ is applied. (Note that in general, $Del_r$ and $Cre_r$ are not graphs, but just fragments.)

$B_{Del} \subseteq L$ is the boundary graph for $Del_r$.

The boundary graph of a fragment $F \subseteq G$ is the smallest graph $B \subseteq G$ completing $F$ to a graph.

$B_{Cre} \subseteq R$ is the boundary graph for $Cre_r$.

The left-hand side of a rule $r$ can have several matches (“occurrences”) in a model $M$. A match $m$ is an injective mapping $m : L \rightarrow M$ assigning a concrete value to each declared variable. $r$ is applicable at match $m$ if $m$ can be extended such that each $pac \in PAC_r$ can be matched and no $nac \in NAC_r$ can be matched.

The effects of applying an applicable rule using match $m$ in $M$ can be described as follows (a full treatment can be found in [10]):

1. The fragment $m(Del_r) \subseteq M$ is deleted from $M$.
2. The fragment $Cre_r$ is inserted into $M$ as a fresh copy and connected with $m(Cxt_r)$.
3. Attribute values are changed according to defined expressions.

In order to guarantee that an application of $r$ is free of side effects, we additionally require the so-called dangling condition of the Double-Pushout Approach to algebraic graph transformation to be fulfilled; each edge in $M$, which is incident to a deleted node in $m(Del_r)$, must have an origin in $Del_r$. This means that the context of a node which is to be deleted must be fully specified; dangling edges are not deleted implicitly (as opposed to the Single-Pushout Approach to graph transformation).

$M_1 \xrightarrow{r,m,n} M_2$ is used as a notation to express that model $M_1$ is transformed into model $M_2$ by applying rule $r$ using match $m$. Co-match $n : R \rightarrow M_2$ shows how the right-hand side is part of model $M_2$.

An application condition $c \in PAC_r \cup NAC_r$ can be translated to a postcondition of the same rule $r$ if its boundary graph $B_c$ is preserved by rule $r$ [10]. The translation can be considered as a normal rule application to $c \cup B_c$.

As an edit rule precisely specifies and implements an edit operation, an edit rule application represents the invocation of an edit operation. The terms edit operation and edit rule as well as the terms edit rule application and edit operation invocation are used as synonyms in the remainder of this paper.

4 Differencing of Models

State-based differencing of two models $A$ and $B$ starts by looking for the “same parts” in $A$ and $B$ (s. Figure 2). There are various methods to identify model elements $a \in A$, $b \in B$ which are considered “the same”. For example, equal unique identifiers are often used as
4 DIFFERENCING OF MODELS

Figure 4: Data structures used to represent differences, edit scripts and edit operations
criterion. Such a pair (a,b) is called a correspondence, and a and b are said to correspond
to each other. A matching between models A and B is a set of correspondences between
elements of A and B.

Given a matching, one can derive a directed delta from A to B as follows: Each model
element of A (or B) not involved in a correspondence leads to a change action which deletes
(or creates) this element. Each non-identical property of two corresponding elements yields
a change action overwriting this property with the value in model B. Each change action
in such a directed delta corresponds to a low-level change which can be observed between
the models (the modification could actually have been caused in a different way).

State-based differencing algorithms use a data structure which implements both a
matching between models A and B and the directed delta from A to B described above.
We refer to this data structure as low-level difference \( \delta(A, B) \). Our schema for the EMF-
based representation of low-level differences is introduced in [16]. It is shown in the upper
right part of Figure 4. Parts of the low-level difference of our running example, namely
two correspondences and two low-level changes, are shown in the lower part of Figure 5.

If we execute one edit operation then the resulting low-level difference can contain
Recognition of Edit Operations

In this section, we present our approach to recognize executions of edit operations in a given low-level difference (see differencing tool chain in Fig. 2), which serves as the basis for the generation of edit scripts as described in Section 6. The basic idea is to group low-level changes according to characteristic change patterns which can be found on the low-level difference. Thus, the set of low-level changes is partitioned into disjoint subsets, each subset containing the changes belonging to exactly one edit operation invocation. These subsets, to which we refer to as semantic change sets, must be disjoint since each low-level change results from the application of exactly one edit operation. The data structure which is used to represent semantic change sets is shown by the upper left part of Figure 4.

Our approach uses so-called recognition rules and applies them to model differences. Recognition rules are introduced in Section 5.1, our rule application strategy is briefly summarized in Section 5.2. Principal restrictions of the approach which are caused by so-called transient effects are discussed in Section 5.3.

Note that the basic idea to recognize edit operations using generated recognition rules is presented in [16]. In this work, it has been significantly extended by pre- and postcondition checks and by establishing trace links between edit rule objects and related recognition rule objects. Trace links are required for retrieving operation arguments and for creating dependency relations between operation invocations as explained in Section 6.

5.1 Recognition Rules

The execution of an edit operation leads to a characteristic pattern of change actions in a low-level model difference. To recognize such a change pattern we use Henshin transformation rules

- to specify change patterns that have to be recognized in a low-level difference,
- to specify application conditions for checking whether pre- and postconditions of a potential edit operation invocation are fulfilled, and
- to specify how to group low-level changes in a change pattern.

These transformations rules are referred to as recognition rules. They can be automatically generated from their corresponding edit rules. In contrast to edit rules, which
A recognition rule which recognizes executions of the refactoring pullUpAttribute is shown as an example in the middle of Figure 5, s. rule recognitionR.pullUpAttribute. It illustrates how context, pre- and postconditions, and change actions of an edit rule are translated into the respective recognition rule:

- The context is translated as follows: Boundary objects to reference deletions must occur in model version A, but not necessarily in version B. Boundary objects to reference creations must occur in model version B, but not necessarily in version A. Preserved objects which are boundary to reference deletions and creations must occur in model versions A and B and have to be corresponding.

- Edit rule preconditions must be fulfilled on model version A, postconditions must
be fulfilled on model version B. In case of edit rule application conditions which can be interpreted as pre- or postconditions, the concrete interpretation must be configured to meet the desired user preferences. In our example, both application conditions are interpreted as postconditions. A pullUpAttribute operation shall be detected only if class cg, to which the attribute p is moved, is still a superclass of cs.

- Finally, a change pattern has to be found by a recognition rule for each change action, i.e. for each model element created or deleted and each value changed by the corresponding edit rule. All effects are declaratively described by our low-level difference representation.

Trace links are established between edit rule objects and their respective recognition rule counterparts. They are indicated by dotted arrows in Figure 5. All trace links are managed persistently.

The lower part of Figure 5 shows how the recognition rule recognitionR_{pullUpAttribute} is matched to the low-level difference $\delta(A, B)$ of our running example. The application of recognitionR_{pullUpAttribute} creates a semantic change set which groups all low-level changes caused by the execution of the edit operation pullUpAttribute.

### 5.2 Rule Application Strategy

Besides the creation of a semantic change set, the only function of a recognition rule is to identify the change patterns which are specific to the type of edit operation which is represented by the semantic change set. Thus, in contrast to their edit rule counterparts, recognition rules are independent of each other and can be applied to the low-level difference representation at all possible matches in parallel.

As long as edit rule applications do not have transient effects i.e., effects that do not result in change actions (see Section 5.3 for more details), all edit rules that have been applied are recognized, even if pairs of rule applications are sequentially dependent. In our example, the application of the edit rule pullUpAttribute sequentially depends on createGeneralization. However, the corresponding recognition rules can be applied in parallel.

In principle, the parallel application of all recognition rules to a given difference can lead to too many semantic change sets which overlap partly or completely. The initial set of potential change sets is then post-processed to find an optimal set partitioning (for details, we refer to [16]).

### 5.3 Restrictions of the Approach

Our approach to the recognition of edit operations assumes that the low-level changes are already computed. Thus, the detection algorithm is efficient in the sense that it runs without backtracking. However, the information which can be exploited by the operation detection is “restricted” to the information which is provided by a low-level difference.

---

1Two change patterns are omitted for recognitionR_{pullUpAttribute} in Figure 5 for the sake of readability, namely the removal and addition of the class references being opposite to their respective ownedAttribute references.
namely the model states A and B and the low-level changes which are observable based on these model versions and the set of given correspondences (see Section 4).

Section 5.3.1 clarifies our notion of a transient effect, the resulting restrictions w.r.t. to the recognition of edit operations are described in Section 5.3.2. In Section 5.3.3 we discuss these principal restrictions with respect to their practical impact.

5.3.1 Transient Effects

If a sequence of operation invocations was applied to a model and one operation invocation removes effects of an earlier one, this causes two types of transient effects; transient elements and transient change actions.

**Transient elements.** Transient elements are model elements, i.e. objects and references of an ASG, that are neither contained in model version A, nor in model version B of a difference $\delta(A, B)$. We further distinguish two kinds of transient elements:

- **Positive transient elements** are created by one operation invocation and deleted in a subsequent edit step of an editing process.
- **Negative transient elements** are deleted by one operation invocation and re-created in a subsequent edit step.

More formally: Given a model $M$ being modified by a sequence of operation invocations implemented as rule applications, then a sequential pair of rule applications $t_1 = M_1 \xRightarrow{r_1} M_2$ and $t_2 = M_2 \xRightarrow{r_2} M_3$ leads to positive transient elements if $n_1(Cre_{r_1}) \cap m_2(Del_{r_2}) \neq \emptyset$. The sequential application of $t_1$ and $t_2$ leads to negative transient elements if $m_1(Del_{r_1}) \cap n_2(Cre_{r_2}) \neq \emptyset$.

**Transient change actions.** Transient change actions are change actions that do not appear as low-level changes in a difference at all, i.e. they cannot be observed in a difference (cf. Section 4).

5.3.2 Consequences

Let us assume a model $M$ being modified by a sequence of edit rule applications $t_1 = M_1 \xRightarrow{r_1} M_2$, $t_2 = M_2 \xRightarrow{r_2} M_3$ and $t_3 = M_3 \xRightarrow{r_3} M_4$.

Let us further assume that an effect of $t_1$ causes an application condition of $t_2$ to be fullfilled, while this effect is later on removed by rule application $t_3$. In such a case, none of the three rule applications $t_1$, $t_2$ and $t_3$ will be recognized by our operation detection approach.

The rule applications $t_1$ and $t_3$ can not be recognized due to transient change actions, regardless of whether they cause positive or negative transient elements; a recognition rule finds a match only if the difference contains all change actions specified by the corresponding edit rule.

Rule applications $t_2$ can not be recognized because one of its application conditions is only transiently fullfilled. A recognition rule, however, finds a match only if all application
conditions are fulfilled either on model version A, or on model version B; application conditions that are to be interpreted as preconditions must be fulfilled on model version A, application conditions that are to be interpreted as postconditions must be fulfilled on model version B (cf. Section 5.1).

To be more precise: If \( t_1 \) and \( t_3 \) cause positive transient elements that are required to extend \( m_2 \) such such that a \( \text{pac} \in \text{PAC}_{t_2} \) can be matched, then this \( \text{pac} \) is only transiently fulfilled. If \( t_1 \) and \( t_3 \) cause negative transient elements that are required to extend \( m_2 \) such such that a \( \text{nac} \in \text{NAC}_{t_2} \) can not be matched, then this \( \text{nac} \) is only transiently fulfilled.

5.3.3 Discussion

Transient change actions which are not observable in a difference are welcome when an operation invocation is taken back completely, e.g., when a model element was created and later deleted, i.e. when a typical \textit{undo} command was executed within an edit environment.

It is not desired, and not harmful either, if complex operation invocations interfere; in such situations, the compound effect can typically be represented by an alternative differences, notably a difference using basic editing commands. Assume, for example, a three level class hierarchy (similar to our introductory example) as shown in Figure 6). Starting from the base version A, we first pull up attribute name from class Developer to class Person (as in the introductory example). Secondly, the attribute is pulled up from Person to Entity. The second operation invocation leads to a transient change action, namely the creation of the containment reference from class Person to the attribute name.
which is not represented in the low-level difference \( \delta(A, B) \). Thus, none of the two applied refactorings is recognized. However, the same effect can also be described by one single attribute movement which directly shifts the attribute name from class Developer to class Entity; this is also a consistency-preserving operation.

From an editing point of view, we call such a transient change action avoidable, because the same effect can be reached without causing transient change actions. From an operation recognition point of view, if such an edit rule is also available, we call the transient effect erasable.

![Diagram showing the editing sequence with transiently fulfilled application condition](image)

**Figure 7: Example of a transiently fulfilled application condition**

Transiently fulfilled application conditions are usually a good hint to not report the respective operation invocation which is missed due to the transient effect. Consider the example shown in Figure 7. On the right hand side of the picture, we can see an editing sequence which yields model version B and which causes transiently fulfilled application conditions; firstly, we introduce generalization relationships such that Person becomes the common superclass of Developer and Manager. Secondly, the common attribute name is pulled up. The generalization relationships are finally deleted.

Obviously, the operation contract for the refactoring `pullUpAttribute` is only fulfilled transiently on model version A'. It is neither fulfilled on model version A, nor on model version B. Thus, an alterantive set of edit operations, for example an attribute movement together with an attribute deletion is reported by the operation detection algorithm.\(^2\)

\(^2\)In fact, the set of given correspondences which edit operations are actually detected in this example. If none of the two name attributes is in a correspondence relationship, two attribute deletions and one creation are reported.
Thus, the transient effect is erasable. Anyway, it is debatable whether the `pullUpAttribute` operation shall be detected at all. Arguably, all application conditions of its operation contract should be still fullfilled on model version B. In this case, the refactoring would be detected by the operation recognition (provided that a proper implementation is available in the given set of edit rules).

To sum up, transient effects are usually not harmful w.r.t. to our approach to edit operation recognition, provided that the effect of an operation sequence causing transient effects can be summarized by an alternative editing sequence. This is not absolutely necessary if differences are lifted for the sake of understandability, e.g. when certain evolution steps must be documented. However, if we must guarantee a complete lifting of low-level differences, as it is the case for the generation of edit scripts (see Section 6), the edit rule set for a given language must be properly designed (see Section 7).

## 6 Generation of Edit Scripts

Section 5 addressed the question how to recognize executions of edit operations in a given low-level difference. From this information, edit scripts can be generated in two subsequent processing steps of the differencing tool chain presented in Fig. 2. The retrieval of actual edit operation parameters is described in Section 6.1, dependency analysis is considered in Section 6.2.

A **consistency-preserving edit script** \( \Delta(A, B) \) is an executable specification showing how to transform a model A into a model B using consistency-preserving edit steps. An edit script is a first class artifact which must be represented in a suitable way. The conceptual data structure which is used to represent operation invocations, their invocation arguments and dependencies between operation invocations is shown by the upper left part of Figure 4. Different options to synthesize edit scripts are discussed in Section 6.3.

### 6.1 Retrieval of Actual Parameters

After the operation detection phase the intermediate result must be further processed to become an executable edit script. Each semantic change set identified represents an invocation of an edit operation. Next, a complete operation invocation must be constructed from each semantic change set. For our running example, two operation invocations, namely `createGeneralization` and `pullUpAttribute`, are created (s. Figure 8).

Subsequently, the parameter retrieval phase has to bind actual parameters, i.e. operation arguments, to each formal parameter declared by the invoked edit operations.

Value parameters can be retrieved by analyzing a match of the recognition rule. Each concrete value can be located in the revised model; it is bound to the respective recognition rule parameter. In our example, parameter \( n \) is a value parameter. The concrete value is "name". The respective value parameter binding is shown in Figure 8.

In order to retrieve the actual values of object parameters, the occurrences of edit rule nodes in a low-level difference \( \delta(A, B) \) can be utilized. These occurrences are identified by composing two mappings which are illustrated in Figure 5.

1. The trace links, labelled \( t \), between edit rule nodes and recognition rule nodes are created and maintained statically for all pairs of edit and recognition rules.
2. The matches of recognition rule nodes, labelled \(m\), are created during operation detection, i.e. when recognition rules are applied to a concrete low-level difference.

Thus, edit rule node occurrences in \(\delta(A, B)\) are identified by the mapping \(o = m \circ t\). Two node occurrences, namely \(o(gen) = m(t(gen))\) and \(o(g) = m(t(g))\), are explicitly shown in Figure 8 while the recognition rules themselves are omitted in this figure.

Having these edit rule node occurrences at hand, object parameter bindings can be created for formal parameters, e.g. \(gen\) of \(createGeneralization\) and \(g\) of \(pullUpAttribute\). In the same way, object parameter bindings are created for all other formal parameters of the example rules, namely \(c1, c2,\) and \(p\).

Actual parameters can be used as input and/or output parameters. For example, parameter \(g\) is an input parameter of operation \(pullUpAttribute\). It determines the generalization along which the attribute represented by property \(p\) is to be pulled up. Parameter \(gen\) of \(createGeneralization\) is an output parameter; it returns the created generalization object. A value used as actual output parameter of one operation invocation can also be used as actual input parameter of a second operation invocation. Such a case leads to a sequential dependency between both invocations.
6.2 Dependency Analysis

Sequential dependencies between edit operation invocations which are caused by input and output arguments are obvious. In our example, the invocation of `pullUpAttribute` sequentially depends on the application of `createGeneralization`. However, not all model elements affected by an operation invocation are necessarily provided as operation arguments. Thus, a complete dependency analysis cannot be reduced to the analysis of parameter mappings.

In general, two edit operation invocations depend on one another if they can be applied in one order and not in the other order or lead to a different effect if applied in the reverse order, i.e. they do not commute. Obviously, testing all combinations of pairs of edit operation invocations is infeasible. Instead, we utilize the implementations of edit operations in Henshin in order to provide an efficient dependency analysis.

To reduce the set of candidates for dependencies that have to be checked, all pairs of edit rules are statically analyzed for potential dependencies by using critical pairs [10]. A critical pair is a pair of subsequent rule applications \( S_1^{r_1, m_1, n_1} \Rightarrow S_2^{m_2, n_2} \) and \( S_2^{r_2, m_2, n_2} \Rightarrow S_3 \) that are dependent in a minimal context. Without considering application conditions, a minimal model \( S_2 \) is constructed by overlapping \( R_1 \) (the right-hand side of rule \( r_1 \)) with \( L_2 \) (the left-hand side of rule \( r_2 \)) in type- and structure-compatible way.

Consequently, the following dependencies are possible if rule \( r_2 \) is applied after rule \( r_1 \):

- **create/use**: The application of rule \( r_1 \) produces a model element that is needed by the application of rule \( r_2 \).
- **change/use**: The application of rule \( r_1 \) changes the value of an attribute that is used by the match of \( r_2 \) including its PACs.
- **delete/forbid**: The application of rule \( r_1 \) deletes a model element that a NAC of \( r_2 \) forbids.
- **change/forbid**: The application of rule \( r_1 \) changes the value of an attribute that is checked by a NAC of \( r_2 \).

In addition, the application of rule \( r_2 \) can have effects that would prevent \( r_1 \) from being successfully applied after \( r_2 \). The resulting kinds of dependencies are the following:

- **use/delete**: The application of rule \( r_2 \) deletes a model element that is needed by the application of rule \( r_1 \).
- **use/change**: The application of rule \( r_2 \) changes the value of an attribute that is used by the match of \( r_1 \) including its PACs.
- **forbid/create**: The application of rule \( r_2 \) creates a model element that a NAC of \( r_1 \) forbids.
- **forbid/change**: The application of rule \( r_2 \) changes the value of an attribute that is checked by a NAC of \( r_1 \).

See Table 6.2 for an overview of all kinds of dependencies. Note that the arrow between rules means that the first rule is depend on the second one. All these different kinds of
dependencies define partial mappings between rules \( r_1 \) and \( r_2 \) in the sense that a model element is mapped to another one if the first one is dependent on the second one. These partial mappings are called potential dependency mappings, since they do not necessarily occur in actual rule applications.

In our example, a potential dependency of kind \textit{create}/\textit{use} is indicated by a dashed arrow in Figure 8; the generalization \textit{gen} has to be created first, potentially by an application of \textit{createGeneralization}, before it can be used as required generalization \textit{g} by \textit{pullUpAttribute}.

<table>
<thead>
<tr>
<th>Dep.</th>
<th>Context/PAC</th>
<th>NAC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes/Edges</td>
<td>Attributes</td>
</tr>
<tr>
<td>( r_2 \rightarrow r_1 )</td>
<td>create/use</td>
<td>change/use</td>
</tr>
<tr>
<td>( r_1 \rightarrow r_2 )</td>
<td>use/delete</td>
<td>use/change</td>
</tr>
</tbody>
</table>

Table 1: Kinds of dependencies

The set of potential dependencies allows us to analyze actual dependencies of operation invocations efficiently. In general, a potential dependency between a critical pair of rules is an actual one between the applications of these rules if the minimal rule application of a critical pair can be embedded into the actual model changes: Given two rule applications \( M_1 \xRightarrow{r_1,m_1,n_1} M_2 \) and \( M_2 \xRightarrow{r_2,m_2,n_2} M_3 \), they are actually dependent if there are at least two elements \( e_1 \in r_1 \) and \( e_2 \in r_2 \) such that there is a potential dependency mapping from \( e_2 \) to \( e_1 \) and \( m_1(e_1) = m_2(e_2) \).

In order to test this condition we utilize again the occurrences of edit rule nodes in the difference of models A and B. In our example, the applications of \textit{createGeneralization} and \textit{pullUpAttribute} are actually dependent because \( \text{gen} \) of \textit{createGeneralization} and \( \text{g} \) of \textit{pullUpAttribute} are mapped to the same model element, i.e. \( o(\text{gen}) = o(\text{g}) \).

6.3 Representation of Edit Scripts

Finally, the obtained edit script must be synthesized and represented in a suitable way.

One option is to serialize the internal EMF representation of the difference data structure which is shown in Figure 8. The corresponding Ecore schema is shown in Figure 4. This option has the advantage that the EMF built-in persistence storage can be used to save and to later load edit scripts. Moreover, edit scripts are synthesized without losing information which might be useful for later use in a patch or merge tool.

Another option is to use a textual notation as shown in Figure 9 for our running example. A sequence of sets of operation invocations represents the partial order induced by the dependency relations of the edit script. Objects used as arguments of input parameters must be identified uniquely by some suitable data value. Note that output arguments are omitted in the notation in Figure 9.

7 Correctness of the Set of Edit Rules

Edit scripts generated with our approach are correct only if all change actions observed in a low-level difference are grouped to semantic change sets during operation recognition.


\[ \Delta(A, B): \]
\{createGeneralization("Developer", "Person")\}
\{pullUpAttribute("name", "Developer → Person")\}

Figure 9: Edit script \( \Delta(A, B) \) for our running example synthesized in a textual notation

To this end, the set of edit rules must be complete in the sense that every consistency-preserving modification of a model can be expressed using edit rules available in this set. Section 7.1 identifies the set of mandatory edit rules; these rules must be provided in order to guarantee that correct edit scripts are generated. Section 7.2 introduces an automated approach to generate a complete set of mandatory edit rules for a given meta-model. Our method of operation recognition additionally requires any modification to be expressible without causing transient effects (see Section 5.3). In Section 7.3, we show that the set of mandatory edit rules is properly designed to meet this additional correctness constraint. Extensions to this set of generated edit rules are discussed in Section 7.4.

7.1 Mandatory Edit Rules

In a first approximation, we can define a complete set of edit rules as follows. For every valid model \( M \) of a given modeling language the following two conditions must hold:

(a) It must be possible to construct \( M \) starting from the empty model \( \emptyset \) by exclusively using creation rules. A creation rule \( r \) is an edit rule which is implemented by creation actions only, i.e. \( Del_r = \emptyset \).

(b) For each creation rule \( r \) there must be an inverse deletion rule \( r^{-1} \) (with \( Cre_r^{-1} = \emptyset \)) such that \( M \) can be reduced to the empty model \( \emptyset \) by exclusively using deletion rules.

While one can define arbitrary many creation and deletion rules for a given modeling language, only these rules which are minimal in the sense that they cannot be split into subsets of change actions which preserve a model’s consistency have to be included in the set of mandatory edit rules.

The set of mandatory edit rules further includes:

(c) changes of attribute values of corresponding elements, i.e. set operations modifying values of object attributes defined by the meta-model;

(d) relocations of model elements if the matcher can produce correspondences between model elements with non-corresponding parents, i.e. move operations that shift a model element to another “position”.

The above mentioned kinds (a)-(d) of operations constitute the finite set of mandatory edit operations which must be implemented by edit rules.
7.2 Generation of Mandatory Edit Rules

To ensure the completeness of the mandatory edit rule set, we choose a constructive approach; mandatory edit rules are generated from the meta-model of the modeling language by using the SiDiff Edit Rule Generator (SERGe) [19].

SERGe is a meta-tool which generates sets for all kinds of mandatory edit rules of a given a meta-model. These sets are complete in the sense that all minimal edit rules are created. Moreover, the generated edit rules comply with the multiplicity constraints of references defined in the meta-model and thus maintain the consistency of a model after application.

Due to multiplicity constraints, minimal creation rules and their inverse deletion rules usually comprise several change actions. An example for such a minimal creation rule which is generated by SERGe is `createGeneralization`, s. Figure 8. According to the UML meta-model, a Generalization object has directly connected mandatory neighbours, i.e. the general and the specific class. Thus, a Generalization object is created together with references to its mandatory neighbours, which have to be provided as input arguments of the edit rule. If a meta-model defines a model element to have mandatory children, such components are created/deleted in the creation/deletion rules generated by SERGe.

7.3 Analysis of Potential Transient Effects

So far, we have only discussed the completeness of the set of mandatory edit rules for a given modeling language. However, our method of operation recognition additionally requires any modification to be expressible without causing transient effects in order to guarantee a complete lifting of low-level differences. We discuss transiently fullfilled application conditions in Section 7.3.1, potential transient change actions are analyzed in Section [7.3.2]

7.3.1 Transiently Fullfilled Application Conditions

Positive and negative application conditions of edit rules can be conceptually devided into two kinds; (a) Consistency checks prevent that a model is transformed into an inconsistent state by the application of an edit operation. (b) The second kind of application conditions are conditions that are part of an operation contract. Complex edit operations such as model refactorings usually specify a comprehensive operation contract. For example, the refactoring operation `pullUpAttribute` can only be applied to an attribute if it is contained by all subclasses of an inheritance hierarchy. Such application conditions are highly specific to the semantics of an edit operation and go far beyond the preservation of the consistency of a model.

Edit operations which are included in our generated edit rule set specify only elementary consistency-checks. As we assume that an edit script transforms a model from one consistent state into another consistent one, application conditions of this kind cannot be fullfilled only transiently. Otherwise, the transformation would yield an inconsistent model, which contradicts our assumption.
7.3.2 Potential Transient Change Actions

Edit scripts are only correct if all change actions are grouped to semantic change sets. Thus, we additionally have to analyze the set of mandatory edit rules w.r.t. potential transient change actions which cannot be recognized by our approach (see Section 5.3). Table 2 shows which rule pairs potentially cause transient change actions when being applied in an editing sequence and which do not. A table entry has to be read as follows; \( r_1 \) and \( r_2 \) are representatives of our four kinds of mandatory edit rules and we assume that \( r_2 \) is applied after \( r_1 \) in an editing sequence.

<table>
<thead>
<tr>
<th>( r_1 )</th>
<th>( r_2 )</th>
<th>creation</th>
<th>deletion</th>
<th>set</th>
<th>move</th>
</tr>
</thead>
<tbody>
<tr>
<td>creation</td>
<td>[1.1]</td>
<td>[1.2]</td>
<td>creation'</td>
<td>[1.3]</td>
<td>creation'</td>
</tr>
<tr>
<td>deletion</td>
<td>[2.1]</td>
<td>[2.2]</td>
<td>[2.3]</td>
<td>[2.4]</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>[3.1]</td>
<td>deletion</td>
<td>[2.2]</td>
<td>set_2</td>
<td>[3.2]</td>
</tr>
<tr>
<td>move</td>
<td>[4.1]</td>
<td>deletion</td>
<td>[2.3]</td>
<td>[4.3]</td>
<td>move_2</td>
</tr>
</tbody>
</table>

Table 2: Analysis of potential transient change actions

Empty entries represent pairs of edit rules \( r_1 \) and \( r_2 \) that do not cause transient effects for all models \( M \) of a given modeling language and for all possible matches \( m_{r_1} : L_{r_1} \rightarrow M \) and \( m_{r_2} : L_{r_2} \rightarrow M \).

Most of the edit rules of a given rule set a pairwise action-independent. Two rules \( r_1 \) and \( r_2 \) are action-independent if each element created by \( r_1 \) cannot be deleted by \( r_2 \) (and vice versa) and if they do not change the value of the same attribute of any element. Rules being action-independent, e.g. rules implementing set and move operations (entries [3.4] and [4.3]) do not cause transient change actions by definition.

All other entries which are empty in Table 2 do not represent rule pairs being action-independent for all representatives of the respective kinds of mandatory edit operations. However, those rule pairs \( r_1 \) and \( r_2 \) that are not action-independent cannot be applied in one editing sequence such that transient change actions occur:

- The application of rule \( r_2 \) sequentially depends on the application of \( r_1 \) in such a way that the effects of \( r_1 \) prevent \( r_2 \) from being successfully applied after \( r_1 \). For example, an element cannot be modified (entries [2.3] and [2.4]) after it has been deleted.

- A pair of rule applications can be in conflict, i.e. only one of these rule applications can be part of an editing process. For example, an element or fragment cannot be created or deleted twice (entries [1.1] and [2.2]).

All those entries not being empty indicate potential transient change actions. In order to demonstrate that set of mandatory edit rules is properly designed, we have to show that each potential transient effect is an erasable one.

Transient change actions are uncritical w.r.t. our approach to operation recognition if the application of \( r_2 \) takes back all effects of the application of \( r_1 \), and the overall effect is like no rule application. This is the case for edit rules being inverse to each other (entries [1.2] and [2.1]). If set and move operations are applied to the same element several times, they are also inverse to each other if they are supplied with proper arguments.
In all other cases, transient change actions are avoidable in the sense that there is an alternative rule application summarizing all effects of applying first $r_1$ and then $r_2$:

- In case of attribute value changes (set) or relocations (move) being followed by a deletion of the respective element or fragment (entries [3.2] and [4.2]), the overall effect can be caused by only applying the deletion operation.

- In case of creating an element or fragment being followed by an attribute value change (set) or relocation (move), the overall effect can be caused by only applying the creation operation, possibly with slightly modified operation arguments (denoted as creation’ in Table 2); Value parameters must be adjusted according to the attribute value change caused by the application of a set operation. Elements which are passed as arguments to object parameters must be selected properly according to relocations of model elements.

- If set and move operations are applied to the same element several times without being inverse to each other, the effect can be summarized by just applying the last attribute value change or relocation of such an editing sequence (entries [3.3] and [4.4] denoted as set$_2$ and move$_2$ in Table 2).

### 7.4 Adapting the Generated Rule Set

Our edit rule generator is not yet capable of interpreting arbitrary well-formedness constraints attached to a given meta-model. In fact, experience shows that model editors do not typically enforce all OCL constraints in a standard, they actually use a simplified meta-model. Thus, each editor environment has its own set of consistency-preserving edit operations and the set of generated edit rules has to be post-processed manually. Typically, some of the generated edit rules have to be merged or complemented by additional application conditions. Such an adaption has to be accomplished carefully: With respect to additional application conditions, only consistency-checks may be added to the generated edit rules (in order to prevent transiently fullfilled application conditions, s. Section 7.3.1). Merged edit rules have to be analyzed w.r.t. potential transient change actions (s. Section 7.3.2). A quantitative assessment of the manual effort needed to adapt the generated edit rules for our case study is given in Section 8.

Obviously, the set of semantically rich complex edit rules which are to be considered useful for a given modeling language has to be engineered manually (see Figure 2). Complex edit rules are implemented by an algorithm which invokes minimal edit rules and which realizes a meaningful editing task. Currently, we support two types of control flows. The sequential composition of edit rules and the composition of edit rules to amalgamation units [3]. An amalgamation unit contains one kernel rule which is embedded into an arbitrary number of multi rules. An amalgamation unit is applied as follows: The kernel rule is applied once. This match is used as a common partial match for each multi rule, which are matched as often as possible. Thus, forall-operations on recurring model patterns can be specified. For example, the effect of the refactoring pullUpAttribute can be achieved by an amalgamation unit; its kernel rule moves one of the “redundant” attributes to the common superclass, one multi rule is needed to delete the remaining attributes from all sibling classes in the inheritance tree.

Complex edit rules are optional extensions of the mandatory rule set and are consistency-preserving by construction.
8 Evaluation

We have evaluated generated edit scripts according to several quality attributes:

Q1: An edit script is correct, i.e. the application of an edit script to model version A finally results in model version B.

Q2: Given a consistent model version A, each edit step of an edit script is consistency-preserving, i.e. each intermediate model as well as model final version B are consistent.

Q3: An edit script raises the level of abstraction, i.e. the algorithm finds as many complex, semantically rich operations as possible.

8.1 Case Study

We have evaluated our approach with respect to quality attributes Q1 to Q3 by an empirical case study.

The test data set of our case study was initially contributed by Herrmannsdörfer et al. [14], and later extended by Langer et al. [25] in order to evaluate some of the contributions of his dissertation (s. Section 9). They studied the evolution of the Eclipse Graphical Modeling Framework (GMF) [12], an open source project that provides a set of generative components and runtime infrastructures for developing graphical editors based on EMF and GEF. The evolution of GMF was observed from release 1.0 to release 2.1 covering a period of two years. In particular, they extracted the revision history of three Ecore models defined by GMF, namely the Graphical Definition Metamodel (gmfgraph), the Generator Metamodel (gmfgen), and the Mappings Metamodel (gmfmap).

Firstly, they checked-out each model version between releases 1.0 and 2.1 from the GMF Subversion repository. Secondly, they manually reverse engineered the edit operations that presumably have been applied between the revisions. The range of available edit operations included simple atomic operations as well as complex refactoring operations known from object-oriented programming. Additionally, model elements were assigned persistent identifiers according to the performed edit steps.

The GMF case study perfectly meets our requirements for the following reasons:

• It is the only case study, to the best of our knowledge, that provides a publicly accessible, extensive evolution history of EMF-based models serving as realistic real-world test data.

• It provides information on which edit steps can be used to correctly specify the model evolution, although the modification could actually have been caused in a different way (which is most likely the case).

• Models elements have universally unique identifiers (UUIDs), thus reliable correspondences can be established in the matching phase of the differencing pipeline.
8.2 Evaluation Setup

Our approach is fully implemented within the SiLift tool environment [17, 33], which we have configured for our case study as follows: The UUID matcher provided by SiLift was used in the matching phase of the differencing pipeline. The edit operation detection engine has been configured with two sets of edit rules, implementing the mandatory and optional edit rules, respectively. In sum, we identified 148 mandatory edit rules available for Ecore models, 134 of them could be fully generated by our edit rule generator. 14 mandatory edit rules had to be manually engineered, most of them are only slightly adapted versions of the edit rules which were initially created by the edit rule generator. Additionally, all of the 32 complex edit operations given in [25] have been implemented manually.

Each pair of successive model revisions \( r_n \rightarrow r_{n+1} \) from the histories of the considered GMF models provides an evolution scenario for which an edit script \( \Delta(M_n, M_{n+1}) \) has been extracted. Basic properties of each evolution scenario are shown by column two and three of Table 8.3. \( |D| \) denotes the number of low-level changes in the difference \( \delta(M_n, M_{n+1}) \), the number of operation invocations contained by the extracted edit script \( \Delta(M_n, M_{n+1}) \) is shown by column \#ops., which further distinguishes mandatory and optional complex operations (\#mandatory/\#complex). In case of the gmfgen history, the 108 evolution scenarios are summarized by average values.

8.3 Correctness of Edit Scripts (Q1)

In order to assess quality attribute Q1, each edit script \( \Delta(M_n, M_{n+1}) \) was applied to its origin model \( M_n \). The result of the application, called \( M_n' \), was compared to \( M_{n+1} \), expecting equal models \( M_n' \) and \( M_{n+1} \).

As shown by column res. of Table 8.3, the edit scripts which were created in our case study are correct for all evolution scenarios, except for the evolution scenarios mappings 1.45 \( \rightarrow \) 1.46 and gmfgren 1.141 \( \rightarrow \) 1.142. Both cases can be ignored because they are caused by an implementation bug in the JavaScript engine which is used by Henshin 0.8.x to modify attribute values.

8.4 Consistency-preservation of Edit Steps (Q2)

The consistency-preservation of edit steps was evaluated by applying each edit script \( \Delta(M_n, M_{n+1}) \) to its origin model \( M_n \) for all evolution scenarios. After each edit step, the resulting state of the model was checked for consistency violations by applying the EMF validation rules provided by Ecore. The obvious expectation was states to be consistent after each edit step.

Somewhat surprisingly, validation errors are already reported for original model versions of the observed histories, i.e. before any edit script was applied. Thus, column \( \delta(\#valid.) \) of Table 8.3 reports the number of changes to the total amount of validation errors which are observed for a model. For example, the initial version of gmfmap violates the Ecore constraint that “a class that is an interface must also be abstract” 9 times. All of these invariant violations are corrected from revision 1.49 to 1.50. Similar changes to the total amount of validation errors can be observed for gmfgren and gmfgren, none of these
changes report an increase of validation errors which results from the application of an edit operation.

### 8.5 Abstraction-level of Edit Scripts (Q3)

Quality attribute Q3 can be directly measured within the edit scripts.

Firstly, the compression factor (compr.), which has been initially introduced in [16], measures how many change actions of a low-level difference are, on average, subsumed by an edit step in the edit scripts. It gives an impression of how the size of a model difference
can be reduced by lifting to the abstraction level of edit operations. We observed average compression factors between 2.28 and 2.59 for the evolving GMF models.

In fact, we have addressed the understandability of model differences for UML class diagrams and Matlab/Simulink models in [27]. The controlled experiments reported even higher compression factors, mainly due to the large amount of redundancies defined by the UML meta-model.

Secondly, we are interested in the recall, i.e. which fraction of the “actually occurred” complex edit operations was detected with our approach. The reference values were given by the manually reverse engineered GMF model histories.

We observed average recall values between 0.61 and 1.00, i.e. some complex edit operation invocations were not found. This is due to the fact that their effect (or application condition) was only visible (or fullfilled) in a transient intermediate state of the actual model versions which have been compared, but not observable in the low-level difference (see Section 5.3). Instead, one or several operations which summarize the effect of a complex one are reported.

An example transient change action can be found in the gmfgen evolution scenario 1.229 → 1.230 as illustrated by Figure 10. Here, the manually reverse engineered edit sequence reports two edit operations which have been applied to attribute requiredPluginIDs of class GenExpressionInterpreter. Firstly, it is “pulled up” to the superclass GenExpressionProviderBase. Subsequently, it is moved to the neighbour class GenExpressionProviderContainer, which is related to GenExpressionProviderBase via reference providers; the respective edit operation is called “collectAttribute” in [14]. Obviously, the affected attribute requiredPluginIDs was owned by class GenExpressionProviderBase only in a transient intermediate state 1.229’ between the revisions 1.229 and 1.230. Consequently, the refactoring sequence “pullUpAttribute”;“collectAttribute” is not recognized. Instead, a simple direct movement of attribute requiredPluginIDs from class GenExpressionInterpreter to GenExpressionProviderContainer is recognized. Thus, the transient effect is erased by the direct attribute movement. In fact, the same observation was made by Langer et al. [25] who present a related approach to the detection of composite edit operations on a low-level difference.

The example shows that recall values have to be interpreted with caution. In this case, the manually reversed sequence of refactorings provides valuable information for the use case of the original case study [14], i.e. the automation of model migration in response to metamodel adaptation. With respect to model versioning, however, difference tool users most likely prefer the change to be explained as simple attribute movement.

As we have shown in test series Q1 and Q2, correctness and consistency-preservation of the extracted edit scripts were not affected by unrecognized complex operations, i.e. we can conclude that all transient effects were erasable ones. Analogously, the precision of our approach can be deduced from the assessments of Q1 and Q2: false positives are shown to not occur because otherwise the target model would not have been correctly reconstructed.
Figure 10: Transient change action in gmfgen evolution scenario 1.229 → 1.230

8.6 Threats to Validity

A threat to validity is that the measured recall values might be misleadingly interpreted w.r.t to the understandability of model differences. The set of expected edit operations for each evolution scenario against which we evaluate our recall of actually detected complex operations is specified manually. In general, there is not only one possible sequence of edit operations which can principally extracted from a given difference. Thus, the set of expected edit operations reflects an individual user’s perception of a difference, which has a strong influence on the reported recall values. In fact, many of the operations that were actually missed due to transient effects would actually not be missed by a large number of developers as the same effect can be achieved by a reduced number of edit steps which do not cause transient effects at all. For instance, as already discussed, it is debatable whether the sequence of refactorings which were manually reverse engineered for gmfgen evolution scenario 1.229 → 1.230 provide a better understanding of what has actually changed than the simple attribute movement which is reported by our extracted edit script.

Another threat to validity is the method used in test series Q1 to check the correctness of edit script applications: the actual result $M_{n'}$ of the application of edit script $\Delta(M_n, M_{n+1})$ to its origin model $M_n$ needs to be compared with the expected result $M_{n+1}$. Obviously, the UUIDs of all model elements in $M_{n'}$ which are created by the application of $\Delta(M_n, M_{n+1})$ differ from the UUIDs assigned to the corresponding elements in $M_{n+1}$. Consequently, meaningful UUIDs are not available in $M_{n'}$ by construction. Thus, we used the similarity-based matching engine of the SiDiff model comparison framework [18] [21] to check the equality of $M_{n'}$ and $M_{n+1}$. Similarity-based matchers can produce correspondences which are generally considered sub-optimal or wrong. However, [39] has analyzed this error for class diagrams and the SiDiff framework; the total number of errors
was typically below 2%.

Moreover, we have not yet been able to conduct a comparable case study for a modeling language different from Ecore due to the lack of fully documented real-world histories of models. However, we see no reason why the approach cannot be successfully adapted to any other modeling language, provided that low-level model differences can be obtained based on reliable correspondences, e.g. provided by UUIDs.

9 Related Work

Many approaches in model versioning are based on logging [6, 13, 26, 31]. These approaches assume closed development environments where one can log user commands in editors and refactoring tools and store them with each model. These logs are just sequences of edit steps, i.e. pure directed deltas: information about dependencies is not available and may be difficult to detect since the semantics of edit operations is often hidden in the source code of model editors.

The problem of how to represent model differences has mainly been addressed in state-based approaches to differencing. Most models of differences proposed so far can only represent elementary model modifications [2, 6, 8, 37].

The grouping of low-level change actions to consistency-preserving edit steps is proposed in a few approaches only [16, 23], and [25]. However, these approaches are not sufficient to find suitable edit scripts because they do not address dependencies between editing steps, the management of non-trivial edit operations, and the identification of operation arguments. Our meta-model is thus a significant extension of previous work.

Langer et al. [25] address the problem of detecting complex operations in differences, however with different goals and assumptions. This approach does not intend to produce executable edit scripts; consequently, the identification of arguments, dependencies between operation invocations etc. are not addressed.

A number of other approaches, e.g. [41], address the reverse engineering of refactorings in the context of mining repositories. The main intention is the same as with [25], namely to document potential instances of refactorings, which shall be manually inspected or just counted. These approaches do not address the production of information required to later re-execute found instances of refactorings.

Könemann [22, 23] presents an approach to model patching which addresses the problem from a broader perspective. This approach aims at understanding and documenting the goals and design decisions manifested by model changes. Könemann proposes a detailed process how a raw difference is transformed into a patch - this process combines several simple heuristics for lifting edit operations - and how the patch is applied to a target model later on. This process resorts several times to interactive interventions of the developer, e.g. in order to semantically enrich the content of the patch, to control the resolution of references, or to correct and to control the effects of the application of the patch. This process is very flexible; in extreme cases, the patch is actually reimplemented for the target model. On the other hand, it is very tedious since each group of related elementary modifications must be identified and treated manually.

In contrast to Könemann’s approach, our approach is much more automated: the lifting of raw differences is fully automated, the application of the patch requires limited
user interaction only. Our approach can exploit additional knowledge on complex edit operations in order to detect arguments in editing steps and to protect against consistency violations.

10 Conclusion

This paper addresses the problem of generating executable differences, called edit scripts, between two models containing all information relevant for executing them. Furthermore, we addressed the problem that edit scripts need to be modified by dropping edit steps in conflict resolution or patching scenarios and that they should remain executable and consistency-preserving even after having been modified.

Most differencing methods currently known produce low-level differences, typically leading to consistency violations in conflict resolution or patching scenarios. Only very few differencing methods for models aim at lifting low-level differences to the level of consistency-preserving edit operations; however, these methods produce differences which provide no information about actual parameters used in edit steps and about dependencies between edit steps.

This paper provides significant extensions compared to previous approaches. It introduces an extended data structure for lifted differences, referred to as consistency-preserving edit script, containing all required information to perform all change actions of a model difference by consistency-preserving edit steps. We also introduced a method and an automated tool chain which assumes that edit operations are implemented as Henshin transformation rules and which automatically generates edit scripts from low-level differences. This method exploits the fact that Henshin transformation rules are formal specifications of edit operations and that the execution of a rule leads to well-defined patterns of change actions. A comprehensive meta-tool environment is provided to adapt the tool chain to arbitrary modeling languages.

In sum, our approach to generate consistency-preserving edit scripts is the only one that supports consistent model patching and merging on a higher abstraction level. In fact, our approach to generate consistency-preserving edit scripts is used in [20] which presents a tool environment for the controlled application of patches. In future work, it can be directly adopted by operation-based merging algorithms as e.g. presented by Lippe et al. [26].

Acknowledgment

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

References

REFERENCES


[6] Bibliography on Comparison and Versioning of Software Models; http://pi.informatik.uni-siegen.de/CVSM


[8] EMF Compare Project; http://www.eclipse.org/emf/compare


REFERENCES


[31] Schneider, Ch.; Zündorf, A.; Niere, J.: CoObRA - a small step for development tools to collaborative environments; ICSE Workshop on Directions in Software Engineering Environments 2004, Edinburgh


[33] SiLift project website; http://pi.informatik.uni-siegen.de/Projekte/SiLift

[34] SysML: OMG Systems Modeling Language; http://www.omgcerml.org


[38] UML: Unified Modeling Language; http://www.uml.org

