A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning

Timo Kehrer, Udo Kelter
Software Engineering Group
University of Siegen, Germany
Email: {kehrer,kelter}@informatik.uni-siegen.de

Gabriele Taentzer
Philipps-Universität Marburg, Germany
Email: taentzer@mathematik.uni-marburg.de

***Abstract*—In model-based software engineering, models are primary artifacts which iteratively evolve and which are often developed in teams. Therefore, comparison and merge tools for models are indispensable. These tools must compare models in a technology-dependent run time representation and will initially derive low-level changes, which can differ considerably from user-level editing commands. Low-level differences are often incomprehensible and should be semantically lifted to the level of editing operations. This transformation of differences depends on the model type, supported editing operations, and user preferences; thus specific transformers are needed, and building them is a challenge. We present a rule-based approach to this problem: low-level differences are represented based on the Eclipse Modeling Framework. They are transformed into representations of editing operations using a rule-based model transformation engine. The necessary transformation rules are automatically derived from basic transformation rules for the editing operations.**

I. Introduction

Model-driven software development (MDSD) is becoming common practice in many application domains. Models are in the center of development activities here, they are iteratively edited during the development process. MDSD requires fully-fledged tool support, which includes tools for comparing, merging, and patching of models, which we will collectively refer to as difference tools.

Currently available difference tools for models are far from satisfying, the low quality of these tools is often considered as one of the biggest obstacles against the wide acceptance of MDSD in practice [2]. This unfortunate situation has one main reason: difference tools must be specifically engineered for each modeling language and, more often than not, be adapted to other tools and user preferences. Methods for developing better difference tools for models with limited implementation effort are therefore a primary concern.

This paper addresses a significant deficiency of current difference tools: they present differences in terms of "low level" changes related to internal, often tool-specific representations of models. In contrast to this, developers perceive models in their external, typically graphical representation, and prefer changes to be explained in terms of basic editing commands and, if available, in terms of refactorings and further complex editing operations.

Difference tools are basically faced with two models which are to be compared and which are represented as contents of persistent storage media or as runtime objects. Persistent storage media include XML files, proprietary file formats, and relational databases. In order to be processed by tools, persistent representations of models must be "loaded", i.e. must be converted into an internal representation consisting of runtime objects. The internal representation can have the same structure as the persistent representation, e.g. some textual format, but often it is converted into a tree structure. It is commonly agreed that comparing textual representations of models does not produce usable results [2], [4], [6] and that models should be compared on the basis of graph-based representations.

Internal representations of models depend on the technologies being used to implement difference tools, notably the specific programming language and, where applicable, modeling frameworks such as the Eclipse Modeling Framework (EMF) [7]. Internal representations can normally be considered as implementations of abstract syntax graphs (ASGs) which represent the conceptual parts of models only.

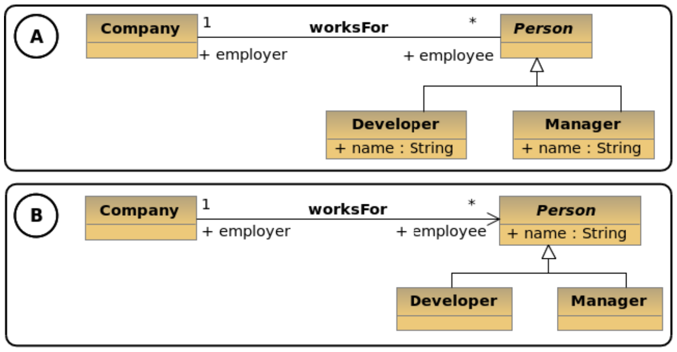

Fig. 1. Original model A and its revision B

The external and internal representations of a model and related changes can differ quite substantially. Figure 1 shows an example based on UML models [30]: Model A is the base version, revision B has been obtained by two editing steps:

1) The navigation of association `worksFor` is restricted to one end, indicated by adding an arrowhead[1].
2) The attribute `name` was moved to superclass `Person` using the refactoring "Pull Up Attribute".

If available difference tools compare the above versions A and B, they will find the following set of low-level changes on the basis of the first editing step:

- Reference `ownedAttribute` from class `Person` to property `employer` has been removed.
- Reference `ownedEnd` from association `worksFor` to property `employer` has been added.
- Reference `class` from property `employer` to class `Person` has been removed.
- Reference `owningAssociation` from property `employer` to association `worksFor` has been added.

These low-level changes are not understandable for normal tool users who are not familiar with meta modeling and the internal representation of models. We will recall the Ecore-representation of UML-models in Section II and present our internal representation of model differences in Section III.

There are several other examples where a seemingly simple change in the external graphical and/or textual representation of model elements causes significant structural changes in the internal representation: dragging an association end to another class or changing the text which represents the multiplicity of an association end, a parameter list, or a list of stereotypes; [15] discusses further examples in state machines and similar types of models.

The second editing step in Figure 1 (pullUpAttribute) leads to a model difference which contains 3 low-level changes: The attribute `name` in the classes `Developer` and `Manager` has been deleted and an attribute `name` has been created in class `Person`. Most likely, the tool user prefers the change being explained as application of refactoring "Pull Up Attribute" on class `Person` for the attribute `name`.

The examples mentioned have in common that a potentially large unstructured set of low-level changes should be grouped in such a way that editing operations can be recognized. Editing operations form an adequate abstraction level for the user perception of model differences. We use the term **semantic lifting** of differences to refer to this transformation of low-level changes to more conceptual descriptions of model modifications.

In this paper, we propose a technique for designing and implementing tool components which can semantically lift model differences.

We assume the usual structure of state-based differencing algorithms as shown in Figure 2: Initially, a matching algorithm identifies corresponding model elements and relations in both models, i.e. corresponding nodes and edges in their ASGs. Model elements and relations not involved in a correspondence are considered to be deleted or created; these insertions and deletions form the low-level difference.

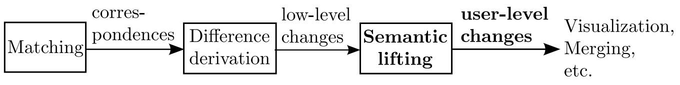

Fig. 2. Model differencing pipeline

The low-level difference is further processed by a "semantic lifting" component which identifies sets of atomic changes that implement an editing operation, and determines these operations as well as their parameters. These results are used by further components for e.g. visualizing differences and merging models, which are out of the scope of this paper.

In our approach, the "semantic lifting" component is implemented using Henshin [1], [13], a transformation engine for EMF models. To that end, a low-level difference is represented as an EMF model. Finding groups of related low-level changes is basically a pattern matching problem, which is solved by the matching engine of Henshin. Such a group is annotated with information about the editing operation it implements by a model transformation.

The main task to implement a semantic lifting component is thus to "program" Henshin rules which find groups of related low-level changes and which annotate these groups accordingly. Most of these Henshin rules can be automatically generated from transformation rules specifying the corresponding editing operations; i.e. in accordance with MDSD principles, we automatically derive these rules rather than to manually program them. The effort required for implementing a semantic lifting component is significantly reduced in this way.

The rest of the paper is structured as follows. Section II recalls how to specify editing operations of EMF models by model transformation rules. In Section III we introduce our representation of model differences which is later extended by the definition of semantic change sets. Section IV introduces our approach to specify instances of semantic change sets by recognition rules, while Section V explains how the recognition rules are to be executed. Section VI evaluates our approach, and Section VII compares our approach with other work. Section VIII concludes the paper.

II. Editing of EMF Models

Throughout this paper, we use the Eclipse Modeling Framework (EMF) as underlying technology for modeling. EMF allows us to specify modeling languages by the Ecore meta-model. Moreover, EMF forms a suitable basis for model editors. For example, graphical editors based on EMF can be generated by the Graphical Modeling Framework (GMF) [11], while the generation of textual editors is well supported by e.g. Xtext [35]. In [28], we show how GMF-generated editors can be extended by complex editing operations specified by EMF model transformation rules. To understand this idea better, we reconsider the example presented in the introduction.

[1]We assume the usual presentation option here that an association without arrowheads is navigable in both directions.

To restrict the navigability of an association to one end, the corresponding edit operation can be specified as in Figure 3. The variable p1 serves as a parameter which shall be bound to the association end property to be restricted in navigability when the rule is applied. Variables r, p2, C1, and C2 will be automatically bound to the association, the opposite association end property and the adjacent classes.

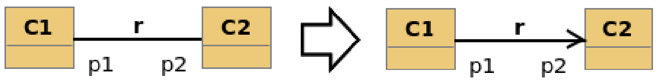

Fig. 3. Sample editing operation in concrete syntax

In order to be processed by modeling tools, editing operations have to be specified based on the internal model representation. Since we consider UML models and take EMF as underlying technology, models are internally represented in the Ecore-based specification of the UML2 syntax. Figure 4 shows a small excerpt from this meta model which is relevant for our example. It shows three model elements Class, Association, and Property together with their various relations. Properties being association ends may be owned by classes (via ownedAttribute) or associations (via ownedEnd). An association end is navigable if it is owned by a class or if it is a navigableOwnedEnd owned by the association. In this paper, we assume navigable ends to be owned by classes and non-navigable ends to be owned by associations.

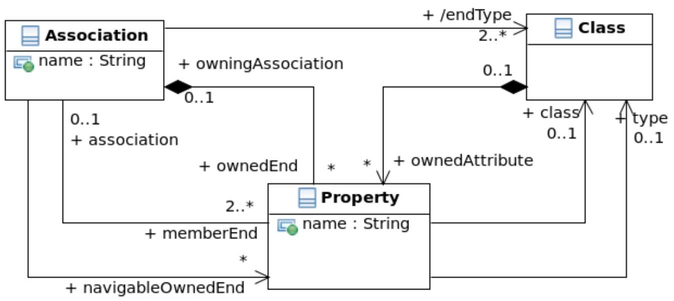

Fig. 4. Excerpt of the Ecore-based implementation of the UML2 superstructure specification

Figure 5 shows the effects of executing the first editing operation in our example, i.e. the restriction of navigability, in the internal model representation. Initially, property employer is owned by class Person, the ownership changes to association worksFor. Thus, references ownedAttribute and class are deleted (shown in red) and references ownedEnd and owningAssociation are created (shown in green).

We use Henshin to precisely specify edit operations by model transformation rules. Henshin supports in-place transformations of EMF models and is based on graph transformation concepts [8]. A Henshin transformation rule can specify model patterns to be found and preserved, to be deleted, to be created, and to be forbidden.

Figure 6 shows Henshin rule editR-restrictNav which specifies the edit operation "restrict navigability". Each

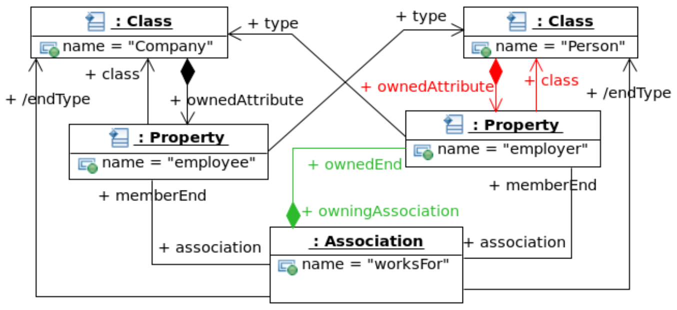

Fig. 5. Internal model modifications after navigability restriction

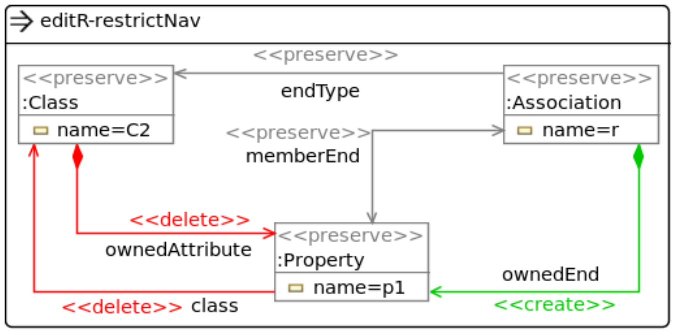

Fig. 6. Edit rule "editR-restrictNav"

EObject and EReference in this model pattern is annotated by a corresponding stereotype. Property p1 (representing the association end to be restricted in navigability), its owning class C2 as well as the association r having p1 as member end have to be found. After the rule application, the matched property has no longer references to class C2, but belongs to the association. Furthermore, a Henshin rule can have variables such as C2, p1, and r. However, p1 is the only input parameter here while the other two are set automatically. Reference pairs that are declared as opposite to each other need to be specified in one direction only in Henshin rules, as the existence of the opposite reference is a general EMF constraint.

III. Model Differences

State-based comparison algorithms compute symmetric differences on the basis of the current states of the models. The computation of a symmetric difference basically consists of two main phases as shown in Figure 2:

1) Initially, a *matching* algorithm identifies corresponding model elements and relations in both models.
2) Subsequently, model elements and relations not involved in a correspondence are determined and a suitable representation of a difference is created. We refer to this creation of a low-level difference as *difference derivation*.

Matching algorithms are out of the scope of this paper, surveys of different approaches can be found in [20], [24].

Figure 7 shows the Ecore-based representation of model differences that is used by the implementation of our approach.

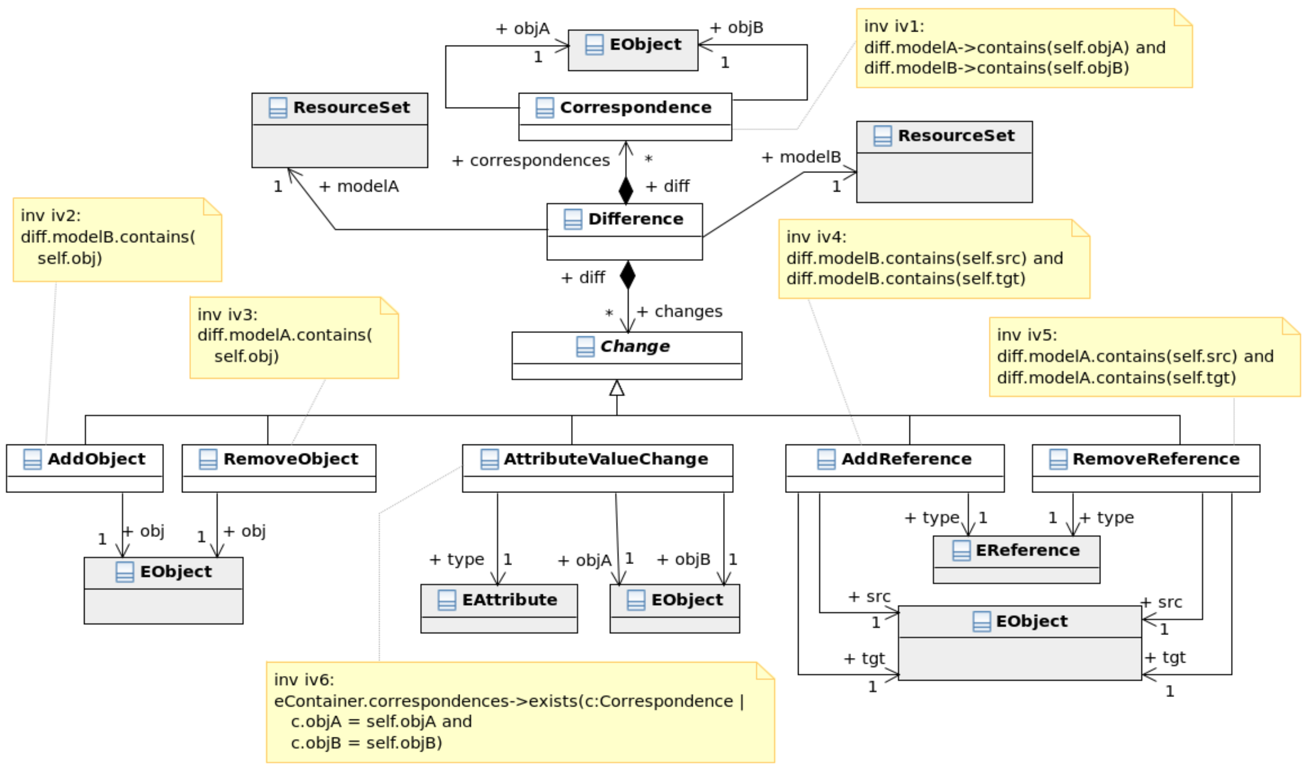

Fig. 7. Ecore-based representation of model differences

Two EMF models A and B that are being compared are represented by ResourceSets. Their difference consists of a set of correspondences representing the common parts of A and B, and a set of changes from model A to model B. A correspondence links an EObject of model A to an EObject of model B (s. invariant iv1). Correspondences between object references are given implicitly by their corresponding source and target objects.

We distinguish the following types of changes:

- An AttributeValueChange represents a value change of an attribute of corresponding objects (s. invariant iv6).
- A change of type AddObject represents the insertion of a new object, i.e. an object that is contained in model B but does not have a corresponding object in model A (s. invariant iv2). Analogously, changes of type AddReference represent references that have been inserted (s. invariant iv4).
- Changes of types RemoveObject and RemoveReference represent the inverse of changes of types AddObject and AddReference, respectively.

We introduce the following notation to describe a change c in a compact manner:

$$c = \langle changeType, context \rangle \tag{III.1}$$

The $changeType$ denotes the type of change, i.e. the concrete subclass of Change, while the $context$ denotes the set of objects (of type EObject or of one of its subtypes EReference and EAttribute) to which the change has been applied.

In terms of our introductory example in Figure 1, the restriction of navigability of the association end employer leads to the following four low-level changes, which were already introduced informally in Section I.

$$\begin{aligned}
\text{rn1} &= \langle "RemoveReference", \\
&\quad \{src = "Person", tgt = "employer", \\
&\quad type = "ownedAttribute"\}\rangle \\
\text{rn2} &= \langle "AddReference", \\
&\quad \{src = "worksFor", tgt = "employer", \\
&\quad type = "ownedEnd"\}\rangle \\
\text{rn3} &= \langle "RemoveReference", \\
&\quad \{src = "employer", tgt = "Person", \\
&\quad type = "class"\}\rangle \\
\text{rn4} &= \langle "AddReference", \\
&\quad \{src = "employer", tgt = "worksFor", \\
&\quad type = "owningAssociation"\}\rangle
\end{aligned}$$

Changes $rn1$ and $rn2$ represent the “move” of property employer from class Person to association worksFor. Changes $rn3$ and $rn4$ are typical examples of conceptually irrelevant pseudo changes [14]; $rn3$ results from the fact that reference class is a redundant information here. Change $rn4$ occurs since ownedEnd and owningAssociation are implemented as mutually reverse (eOpposite) reference types in the Ecore-based UML2 meta model.

Let us now consider the second editing step of our example of Figure 1. Here, the attribute name is moved from subclasses `Developer` and `Manager` to their superclass `Person` by means of the well-known refactoring operation "Pull Up Attribute" [10]. Let us assume that our matching contains correspondences between all classes in A and B having equal names, and a correspondence between the attributes `Manager.name` in A and `Person.name` in B. The following low-level changes are derived then. (We omit several low-level changes which are pseudo changes. They are similar to those which occurred in the first editing step.)

$$\begin{aligned} \text{pua1} &= \langle \text{"}RemoveObject\text{"}, \\ &\quad \{obj = \text{"}Developer.name\text{"}\}\rangle \\ \text{pua2} &= \langle \text{"}RemoveReference\text{"}, \\ &\quad \{src = \text{"}Manager\text{"}, tgt = \text{"}Manager.name\text{"}, \\ &\quad type = \text{"}ownedAttribute\text{"}\}\rangle \\ \text{pua3} &= \langle \text{"}AddReference\text{"}, \\ &\quad \{src = \text{"}Person\text{"}, tgt = \text{"}Person.name\text{"}, \\ &\quad type = \text{"}ownedAttribute\text{"}\}\rangle \end{aligned}$$

Change $pua1$ represents the deletion of the attribute `Developer.name`, while changes $pua2$ and $pua3$ represent the move of attribute `Manager.name` from class `Manager` to class `Person`.

IV. Semantic Change Sets

A. Definition of Semantic Change Sets

Our introductory example in Figure 1 has already shown that user-level edit operations often lead to many low-level changes which are hard to understand for normal users. This problem is further aggravated by the fact that the set of low-level changes can be listed in an arbitrary order and that low-level changes stemming from different edit operations can be mixed randomly. The objective of semantically lifting a model difference is thus to partition the set of low-level changes into disjoint subsets, each subset containing the changes belonging to exactly one edit operation. These subsets must be disjoint since each low-level change results from the application of exactly one edit operation. We call these subsets **semantic change sets**. Formally, we denote the set of all semantic change sets as

$$CS = \{cs_1, ..., cs_m\} \tag{IV.1}$$

with $m \leq |C|$ where C is the overall set of changes. Moreover, we have $cs_i \subseteq C$ for all i, $1 \leq i \leq m$, $\bigcup_{i=1}^{m} cs_i = C$, and $cs_i \cap cs_j = \emptyset$ for all $1 \leq i < j \leq m$.

In our example of Figure 1, the set of all change sets is

$$CS_1 = \{\{rn1, rn2, rn3, rn4\}, \{pua1, pua2, pua3\}\}.$$

The first change set contains the result of the edit operation "restrictNav", while the second one is the result of edit operation "pullUpAttribute"[2].

In order to represent change sets, we must extend the Ecore-based representation of model differences in Figure 7. These extensions are shown in Figure 8. Objects of type `Change` which represent the low-level changes, are grouped by `SemanticChangeSet` objects which represent the result of an edit operation.

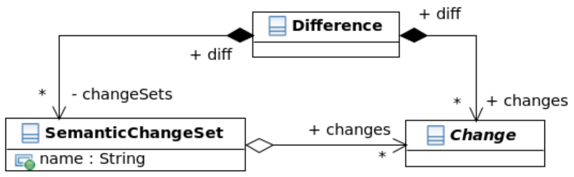

Fig. 8. Difference model extension: Representation of semantic change sets

B. Semantic Change Set Specification

The application of a specific type of edit operation results in a change pattern on our Ecore-based difference representation which is characteristic of this type of edit operation. Thus, we can make use of Henshin transformation rules

1) to specify change patterns that have to be recognized in an Ecore-based difference representation and
2) to specify how to group low-level differences contained in a change pattern.

A transformation rule which handles a specific type of edit operation is referred to as **change set recognition rule**.

As an example, an excerpt of the change set recognition rule "recognitionR-restrictNav" recognizing edit operation "restrictNav" is depicted in Figure 9. Although the concrete identifier names are technically irrelevant in terms of Henshin transformation rules, we use the variable names of Figure 6 to build identifier names. That way objects of the internal model representations in "recognitionR-restrictNav" can be mentally related to those of the edit rule "editR-restrictNav" (s. Figure 6).

As we can see in Figure 9, objects which are preserved by the edit rule are linked by correspondences in the difference representation. Three instances of this *correspondence pattern* can be identified: for association `r`, for class `C2` and for property `p1` which is moved from `r` to `C2`.

No direct correspondence mappings are established between preserved references in the difference representation. Corresponding references are identified by their context, i.e. corresponding source and target objects. In "recognitionR-restrictNav", this *preserved reference pattern* can be found three times: for the reference `memberEnd` from `r` to `p1`, the opposite reference `association` from `p1` to `r`, and the reference `endType` from `r` to `C2`.

Additionally, our sample recognition rule "recognitionR-restrictNav" contains four change patterns (two *remove reference patterns* and two *add reference patterns*) representing changes $rn1$ to $rn4$ (s. Section III). Figure 9 shows patterns for changes $rn1$ and $rn2$ only. They can be easily identified in Figure 9 by means of their respective change types and contexts.

A `SemanticChangeSet` object is created by each application of "recognitionR-restrictNav"; this object groups all related low-level changes. Additionally, the created change set

[2]As already mentioned, we have omitted several pseudo changes here for the sake of readability.

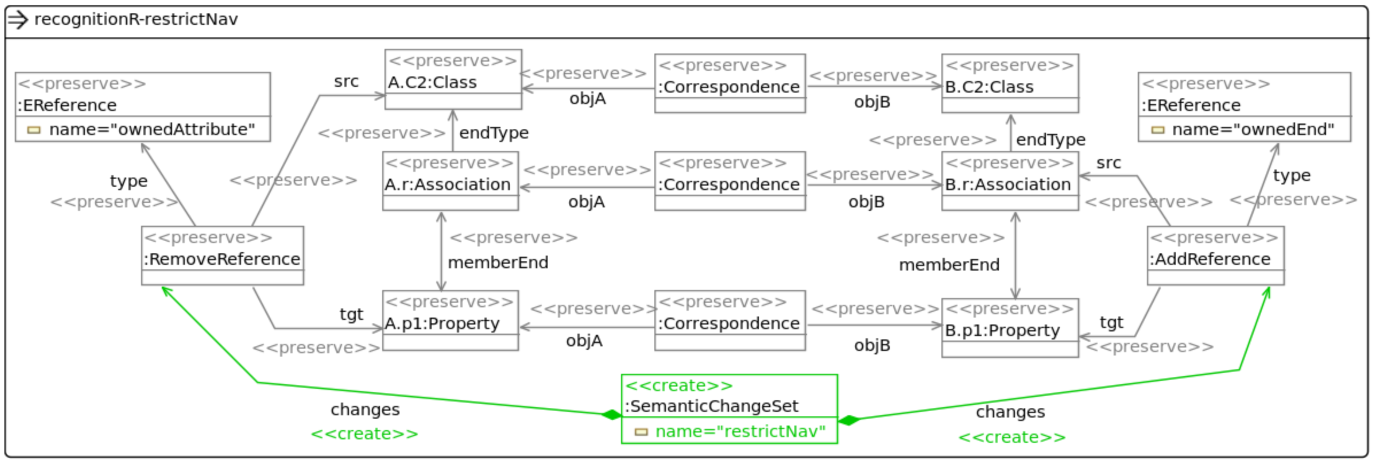

Fig. 9. Sample recognition rule “recognitionR-restrictNav”

will be inserted into the difference representation (which is not shown in Figure 9 due to space limitations).

C. Generation of Recognition Rules

Change set recognition rules are getting complex very quickly. However, they are very schematic and can be automatically generated from their corresponding edit rule. The core of this transformation is presented as function `editR2-RecognR` in the listing below. Let `er` be the edit rule which serves as input parameter and which will be transformed to a corresponding recognition rule.

```
function editR2RecognR(Rule er){
  Rule rr = new Rule();
  createChangeSet(rr);
  for each EObject o ∈ (er.preserved)
    createCorrespondencePattern(rr, o);
  for each EReference r ∈ (er.preserved)
    createPreservedReferencePattern(rr, r);
  for each EObject o ∈ (er.deleted){
    createRemovedObjectPattern(rr, o);
    linkChangeToChangeSet(rr, o);
  }
  for each EReference r ∈ (er.deleted){
    createRemoveReferencePattern(rr, r);
    linkChangeToChangeSet(rr, r);
  }
  for each EObject o ∈ (er.created){
    createAddObjectPattern(rr, o);
    linkChangeToChangeSet(rr, o);
  }
  for each EReference r ∈ (er.created){
    createAddReferencePattern(rr, r);
    linkChangeToChangeSet(rr, r);
  }
  return rr;
}
```

All subroutines creating instances of the respective patterns are not explained in detail here, but can be implemented straightforward according to the explanation of our “recognitionR-restrictNav”-example above. Subroutine `createChangeSet` adds a `SemanticChangeSet` object to the `created`-part of the recognition rule. This `SemanticChangeSet` object is linked with all `Change` objects which are created by change pattern instances; these links are created by the subroutine `linkChangeToChangeSet`.

An automatically generated change set recognition rule has to be manually post-processed only if the corresponding edit rule has side effects which are (a) not explicitly expressed in the edit rule and which (b) cannot be derived from information in the meta model.

In our sample edit rule “editR-restrictNav”, the implicit creation of the `owningAssociation` reference from `p1` to `r` can be derived automatically, as `owningAssociation` is declared to be opposite to reference `ownedEnd` from `r` to `p1`, which is explicitly created.

D. Non-static Change Patterns

So far, we have considered editing operations which result in *static change patterns*, i.e. each editing operation can be specified by one transformation rule. However, there are also editing operations resulting in *non-static change patterns*. The refactoring operation “pullUpAttribute” is an example of this: it is applied to all common attributes in the set of all direct subclasses of a given class. Since the number of common attributes and the number of subclasses can vary, we need a concept to capture this variability.

Although simple model transformation rules as presented in Section II can combine a number of basic model operations into one transaction, the variability described above cannot be specified on that basis. We need an additional concept to express recurring model patterns, which we call *multi-object structures*.

The model transformation system Henshin offers amalgamation units to specify transformations of multi-object structures [1], [3]. An amalgamation unit contains a kernel rule and an arbitrary number of multi rules. A kernel rule is a simple model transformation rule which can be equipped with a set of multi-rules which include the kernel rule. Each multi-rule specifies one multi-object structure and its transformation. 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, multi-object structure transformations are performed as often as corresponding structures occur in a given model. An edit rule for refactoring "pullUpAttribute" is presented in [1]. Its kernel rule moves one of the common attributes which shall be pulled up from a class to its superclass. One multi-rule is needed to delete the common attribute from each further subclass.

We have adopted the amalgamation concept to handle non-static change patterns in recognition rules. To that end, our algorithm for recognition rule generation is extended by the function `editAU2RecognAU`. A sketch of this function is listed below: Parameter `eau` denotes the editing amalgamation unit and variable `rau` denotes the recognition amalgamation unit to be created. The kernel rule of `eau` is transformed to the kernel rule of `rau` according to the function `editR2RecognR()` introduced in section IV-C. In the same way, each multi-rule of `eau` is transformed to a multi-rule of `rau` properly embedding the kernel rule of `rau` into each multi-rule.

```
function editAU2RecognAU(AmalgmUnit eau) {
  AmalgmUnit rau = new AmalgmUnit();
  rau.kernel = editR2RecognR(eau.kernel);
  for each Rule em ∈ eau.multi {
    rau.multi.add(editR2RecognR(em));
    embedKernel(rau, em);
  }
  return rau;
}
```

V. Editing Operation Recognition

A. Rule application strategy

The recognition of an edit operation is based on a set of low-level changes as collected in a difference. Transient model elements, which are created by one edit operation and deleted by a subsequent one, do not appear at all in the resulting difference. A recognition rule, however, will only find a match if the difference contains *all* low-level changes specified by the corresponding edit rule. We assume here, that the effect of each edit operation, that is applied in an editing sequence, is either removed completely by a later operation or it is entirely preserved. Thus, the sequential order in which we apply the change set recognition rules is irrelevant. Additionally, change set recognition rules do not delete elements and thus, are parallel independent, i.e. the recognition rules can be applied to the difference representation at all possible matches in parallel.

However, the above rule application strategy can lead to too many change sets, i.e. there can be false positives. As an example, we extend our set of edit operations defined for UML class models by two further edit operations, namely the edit operation "deleteAttr", which removes an attribute from a class, and the edit operation "moveAttr", which moves an attribute from one class into another. In our running example of Figure 1, the recognition rules for "deleteAttr" and "moveAttr" will find the additional potential change sets $\{pua1\}$ and $\{pua2, pua3\}$, respectively. In sum, all four recognition rules together will find the following change sets:

$$PCS_1 = CS_1 \ \cup \ \{\{pua1\}, \{pua2, pua3\}\},$$

CS_1 (see Section IV-B) contains the change sets which represent the edit operations that have been actually applied. The change sets contained in PCS_1 are not mutually disjoint, therefore some of them must be discarded.

In general, let D be a given difference, i.e. a set of low-level changes, and PCS_D be the set of *potential change sets* which are created by applying all recognition rules on D as often as possible. Thus

$$\forall p \in PCS_D : \ p \subseteq D \tag{V.1}$$

If some potential change sets overlap, i.e.

$$\exists p, q \in PCS_D : \ p \cap q \neq \emptyset, \tag{V.2}$$

then PCS_D must be postprocessed as described in the next section.

B. Postprocessing

The goal of the postprocessing phase is to determine a subset of set PCS_D of potential change sets which conforms to the conditions for sets of semantic change sets (s. Section IV-A). The postprocessing of PCS_D results in a set partitioning problem, which is basically an optimization problem. Due to the lack of a clear optimization criterion, we assume that reporting a minimal number of editing operations reflects the user perception of a model difference most adequately. Thus, we are looking for a set $PCS_{min} \subseteq PCS_D$ of potential change sets such that the following conditions hold for $PCS_{min} = \{p_1, ..., p_k\}$:

- $p_i \cap p_j = \emptyset$, $\forall 1 \leq i < j \leq k$,
- $\bigcup_{i=1}^{k} p_i = D$, and
- k is minimal.

We employ the following heuristics in order to efficiently reduce PCS_D and finally determine PCS_{min}:

Firstly, we are looking for change sets that do not overlap with other change sets. Such change sets are very frequent. They must obviously be included in PCS_{min} and do not have to be dealt with further. In our example, this is the case for change set $\{rn1, rn2, rn3, rn4\}$.

Secondly, we search for each change set p which properly includes smaller change sets $q_1 \dots q_m$ which do not overlap with any other change set not included in p. In this case, change sets $q_1 \dots q_m$ will be discarded and are not included in PCS_{min} because, generally, an edit operation which covers a larger change set is preferred over edit operations which cover a smaller, included set of changes. In our running example, the change set $\{pua1, pua2, pua3\}$ representing the "pullUpAttribute" refactoring will thus be preferred over the smaller change sets $\{pua1\}$ and $\{pua2, pua3\}$, which can be discarded. These cases occur whenever a single operation is composed of a set of smaller operations or a core operation has one or more extensions.

Finally, the remaining change sets are partially overlapping. This reduced set partitioning problem has to be solved by combinatorial optimization. Our practical evaluation has shown that these cases are hypothetically as long as the set of edit operations consists of basic operations as they are offered by typical editors for graphical modeling (s. Section VI).

VI. Evaluation

A. Case Studies

We implemented change set recognition rules for UML class models and Matlab/Simulink block diagrams. These two visual modeling languages are good representatives to show the applicability of the approach. UML is the standard modeling language for software systems and comes along with a comprehensive meta model. Matlab/Simulink [22] is a domain-specific language that is a de-facto standard for modeling embedded systems; the characteristics of Simulink models and the way how they are edited differ significantly from class models.

In the case of UML class models, our set of recognition rules covers 41 edit operations which are taken to a large extent from the atomic rules given in [31]. The recognition rules are based on the EMF implementation of the UML meta model. Most parts of the change set recognition rules are involved in collecting pseudo changes which result from the complex, often redundant structure of the UML meta model. Many change patterns resulting from UML edit operations are structurally similar, therefore we can define a set of rule templates which can be adapted easily to specify concrete UML edit operations. In sum, it took us several days of work only to develop the recognition rules for UML class models.

In the case of Matlab/Simulink, we identified 16 different edit operations which are applicable to block diagrams in terms of the Matlab/Simulink IDE. A meta model for Matlab/Simulink is not publicly available. Therefore we developed an own meta model for Simulink diagrams and implemented it in EMF. We also implemented a parser which converts Simulink models from the proprietary format into their EMF representation. Our Simulink meta model contains only few redundancies; the number of pseudo changes we have to address within the recognition rules is significantly lower than in the case of UML models. The edit operations on Simulink models turn out be more complex and less schematic than those on UML class models. Our rule templates for typical UML edit operations can be reused for 11 of our 16 Simulink edit operations. Although the number of edit operations is much smaller than in case of the UML class models, it took us approximately the same amount of time to implement our Simulink recognition rules.

Editors for graphical modeling typically offer pairs of operations to add and delete model elements of specific types. These pairs of operations are basically inverse to each other, i.e. the effects of an edit operation are removed completely by the inverse operation. In addition, there are some move operations which are inverse to themselves. For both, UML class models and Simulink block diagrams, we were able to specify respective sets of edit operations. Differences being produced by these sets of editing operations therefore adhere to our basic assumptions for edit operations stated in Section V-A.

B. Test Cases

We developed different test scenarios to which we applied our change set recognition for UML class models. Some test cases are synthetic, these pairs of models were manually created by applying predefined sequences of edit operations. In the following tables, we group the synthetic test cases in three categories, the first and second being borderline test cases. Category UML_1 contains pairs of unchanged models, which result in empty differences. Category UML_2 contains pairs of models which have no corresponding model elements at all, i.e. differences are large. Category UML_3 contains pairs of models that were modified by a random selection of a set of edit operations. We create one representative test case for categories UML_1 and UML_2, respectively, and six representatives falling into category UML_3.

The other test cases are taken from real projects. We analysed consecutive pairs of an existing history of 10 revisions of a flight ticketing data model and manually reverse engineered sequences of edit operations which we later used as reference output data for our test cases (UML_4). In the case of Matlab/Simulink, we extracted test data from an existing model repository of an industrial partner from the automotive domain. Here, we analyzed the differences between four snapshots of a block diagram which models the behaviour of a cruise control (SIM).

Table I summarizes characteristics of our test cases. Firstly, an overview of the sizes of compared models is given by the average number of objects (NoO) and the average number of object references (NoR) in terms of the internal representation. Secondly, the measures of two difference metrics, namely the average numbers of correspondences (NoC) and applied edit operations (NoE) indicate the extent to which the compared models were differing from each other.

TABLE I
Test model and difference measures

	avg. NoO	avg. NoR	avg. NoC	avg. NoE
UML_1	45	92	45	0
UML_2	39	73	1	52
UML_3	32	63	26	13
UML_4	118	284	103	20
SIM	4963	6750	4901	59

C. Test Results and Analysis

The first column of Table II shows the average size of the low-level difference (avg. $|D|$), i.e. the number of low-level changes. The other columns list the minimum, maximum, and average value of the *compression factor*, which is an indicator how much a user's perception of a difference is improved. It is defined as follows:

$$cf = \frac{|D|}{|CS|} \quad \text{(VI.1)}$$

where $|CS|$ denotes the number of recognized semantic change sets. The compression factor measures how many low-level changes are grouped by a semantic change set. In the case of UML$_3$, UML$_4$ and SIM, which cover multiple test cases, Table II lists average values for $|D|$ and cf.

TABLE II
DIFFERENCE COMPRESSION THROUGH SEMANTIC LIFTING

	avg. $\lvert D\rvert$	$\min(\frac{\lvert D\rvert}{\lvert CS\rvert})$	$\max(\frac{\lvert D\rvert}{\lvert CS\rvert})$	avg. cf
UML$_1$	0	-	-	-
UML$_2$	596	4	91	11.4
UML$_3$	131	1	64	18.0
UML$_4$	310	1	55	15.7
SIM	484	1	56	6.2

The compression rates show that model differences can in fact be optimized significantly through semantically lifting to user-level edit operations.

In case of the UML models, the high compression rates might be surprising, since most edit operations appear to be quite simple. The high compression rates are mainly caused by the fact that the meta model is far from being optimized for model comparison and leads to a large number of pseudo changes. The maximum compression rates usually occur in the context of delete operations. For instance, the maximum number of 91 low-level changes results from the deletion of a UML class with all its adjacent relations (associations, dependencies etc.) and contained attributes. The minimum compression rates occur in the context of edit operations which change attribute values, e.g. the name of a model element, which leads to one low-level change only.

In the case of Simulink models, the average compression rates are lower, but still significant, and our evaluation shows that it is definitively worth to semantically lift differences here, too. The maximum number of 56 low-level changes in a semantic change set was due to a complex edit operation which extracts an entire subsystem; the manual inspection of the related low-level changes is very tedious.

A basic assumption of our approach is that the low-level difference contains complete change sets which in turn means that the matching engine producing the matching, must be fairly reliable. In the case of UML models, we computed correspondences on the basis of persistent identifiers, which is a very common approach. Matchings computed this way are very reliable. As expected, all test cases lead to a correct recognition of semantic changes sets, i.e. *all* low-level changes are grouped by change sets which represent the edit operations being applied to our test models.

The percentage of correctly recognized change sets is more than 95% in each of the Simulink test cases. There are two reasons why some edit operations are not recognized: (a) Persistent identifiers are not available in the case of Simulink models. Here, we use the similarity-based matching engine of SiDiff [26] for computing correspondences. Due to comparison heuristics, a small part of the change sets differs slightly from the reference edit sequences, and thus is not recognized. (b) In rare cases, which depend on the state of a model, Simulink edit operations have complicated side-effects, e.g. implicit creation of default value specifications. These effects are not directly visible in the user interface, and might even be irrelevant from a user's point of view. However, even in cases where low-level changes are not grouped to change sets, the manual inspection of the resulting lifted difference is much easier than the inspection of the raw low-level difference. Nonetheless, it appears a basic requirement, that the tools, which process semantically lifted differences, must be capable of working with low-level changes and semantic change sets simultaneously. With respect to difference visualization, for example, the clickable list of local changes [16] is a proper concept to support low-level changes and semantic change sets in parallel.

We also measured the execution times for the semantic lifting of the low-level differences. Typically, these times are much smaller than the time needed to compute the low-level difference. In the case of all Simulink test cases, for example, more than 90% of the overall comparison runtime is consumed by the computation of correspondences.

These figures are mainly due to the fact that in practice only a small part of the compared models is changed. In other words, no matter how sizes are measured, model differences are always much smaller than the compared models (see Table I). To sum up, our performance analysis demonstrates that semantic lifting is practicable for interactive model comparison.

VII. Related Work

Difference tools for models have been addressed by a large number of publications recently; the bibliography [5] compiles about 300 publications in this field, most of them dating from 2003 or later.

One class of approaches [12], [21], [23] is based on logging, i.e. editing processes are logged at the level of user commands or lower levels. Thus the problem addressed by this paper disappears. However, logging-based approaches require closed environments and do not work with independently created models, thus they are not a general solution of the problem.

Most approaches are state-based, surveys of them can be found in [9], [20], [24]. Virtually all of these methods consider models as graphs and have a similar processing structure like the basic differencing pipeline shown in Figure 2.

Some of these approaches [32], [34] are specific to certain domains, i.e. adapted to a specific model type. They can specifically handle each type of model elements. These approaches use special matching strategies and difference derivations; they are not general enough for other model types.

Only a few approaches are generic in the sense that they can be adapted to any model type, with varying degree of effort and success [6], [19], [25], [29]. These algorithms assume a standard graph representation such as Ecore, and are typically configured by the meta model of a model type and by additional configuration data. They have in common that they deliver only low-level differences, thus all of them can be used to perform the first two steps of the differencing pipeline shown in Figure 2. We are not aware of a generic algorithm which semantically lifts the low-level differences and which can be adapted to a large variety of model types.

Könemann [17], [18] addresses a similar problem like ours: A difference can be used as a patch which is to be applied to a model. Problems arise if this model differs from the original base model from which the difference was computed. The "literal" difference is transformed into a more "fuzzy" specification of the desired changes, and one attempts to catch the change intention by grouping changes using name patterns or OCL queries. The definition of groups of changes and the later application of the patch rely heavily on manual control and corrections by users; in contrast to that, we aim at an automated transformation of low-level differences. In fact, our method of semantically lifting differences can beneficially be used before transforming the exact (lifted) difference into a "fuzzy" patch.

A number of publications have addressed the reverse engineering of refactorings in the context of mining repositories. For example, [33] proposes to export relevant difference information in a database and to use queries for finding instances of refactorings. Our problem of actually annotating and lifting a difference is not directly addressed. The overhead of setting up a database and creating all required indexes, which is relevant when mining repositories, is a significant performance problem when comparing only two models.

VIII. Conclusion

In this paper, we have addressed the problem of how to semantically lift low-level differences, which are produced by currently available differencing engines for models, towards user-comprehensible specifications of changes. We assume that solutions of this problem are acceptable only if they can be adapted to the specific model type and user preferences, thus solutions must be individually engineered. Our approach uses a standard model transformer (Henshin) for finding instances of editing operations and annotating a low-level difference, and reduces the "programming" of Henshin largely to a transformation of edit rules to recognition rules, which drastically reduces the implementation effort. Our tests have shown that the components implemented in this way performed well with fairly high numbers of editing operations to be recognized.

References

[1] Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems 2010, Oslo; LNCS 6394, Springer; 2010

[2] Bendix, L.; Emanuelsson, P.: Collaborative Work with Software Models - Industrial Experience and Requirements; Proc. Intl. Conf. Model-Based Systems Engineering, Haifa; http://www.mbse-org.org; 2009

[3] Biermann, E.; Ermel, C.; Taentzer, G.: Lifting Parallel Graph Transformation Concepts to Model Transformation based on the Eclipse Modeling Framework; Electronic Communications of the EASST 26; 2010

[4] Proc. Intl. ICSE Workshop on Comparison and Versioning of Software Models, Vancouver; IEEE; 2009

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

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

[7] EMF: Eclipse Modeling Framework; http://www.eclipse.org/emf

[8] Ehrig, H.; Ehrig, K.; Prange, U.; Taentzer, G.: Fundamentals of Algebraic Graph Transformation; Monographs in Theoretical Computer Science, Springer; 2006

[9] Förtsch, S.; Westfechtel, B.: Differencing and Merging of Software Diagrams - State of the Art and Challenges; p.90-99 in: Proc. Intl. Conf. Software and Data Technologies, 2007; INSTICC Press; 2007

[10] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.: Refactoring: Improving the Design of Existing Code; Addison-Wesley; 1999

[11] GMF: Graphical Modeling Framework; http://www.eclipse.org/gmf

[12] Herrmannsdörfer, M.; Kögel, M.: Towards a Generic Operation Recorder for Model Evolution; p.76-81 in: Proc. 1st Intl. Workshop on Model Comparison in Practice, Malaga; ACM; 2010

[13] EMF Henshin Project; http://www.eclipse.org/modeling/emft/henshin

[14] Kelter, U.: Pseudo-Modelldifferenzen und die Phasenabhängigkeit von Metamodellen; p.117-128 in: Proc. Software Engineering 2010, Paderborn; LNI 159, GI; 2010 (available in German only)

[15] Kelter, U.; Schmidt, M.: Comparing state machines; p.1-6 in: Proc. ICSE Workshop on Comparison and Versioning of Software Models, Leipzig; ACM; 2008

[16] Kelter, U.; Schmidt, M.; Wenzel, S.: Architekturen von Differenzwerkzeugen für Modelle; p.155-168 in: Proc. Software Engineering 2008. Munich; LNI 121, GI; 2008 (available in German only)

[17] Könemann, P.: Model-independent differences; p.37-42 in: [4]

[18] Könemann, P.: Capturing the Intention of Model Changes; p.108-122 in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems 2010, Oslo, Part I; LNCS 6394, Springer; 2010

[19] Kolovos, D.S.; Paige, R.F.; Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML); p.215-229 in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems 2006, Genova; LNiCS 4199, Springer; 2006

[20] Kolovos, D.; Ruscio, D.; Pierantonio, A.; Paige, R.F.: Different Models for Model Matching; p.1-6 in: [4]

[21] Lippe, E.; Oosterom, N.: Operation-based Merging; p.78-87 in: Proc. ACM SIGSOFT Symp. Software Development Environments 1992; ACM SIGSOFT Software Eng. Notes 17:5; 1992

[22] Mathworks: Matlab/Simulink; http://www.mathworks.com/simulink

[23] 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

[24] Selonen, P.: A Review of UML Model Comparison Techniques; p.37-51 in: Proc. Nordic Workshop on Model Driven Engineering 2007, Ronneby; U. Göteborg; 2007

[25] Selonen, P.; Kettunen, M.: Metamodel-Based Inference of Inter-Model Correspondence; p.71-80 in: Europ. Conf. Software Maintenance and Reengineering 2007, Amsterdam; IEEE Computer Society; 2007

[26] The SiDiff Project; http://www.sidiff.org

[27] Taentzer, G.; Ermel, C.; Langer, P.; Wimmer, M.: Conflict Detection for Model Versioning Based on Graph Modifications; in: Proc. of Intl. Conf. Graph Transformation; LNCS 6372, Springer; 2010

[28] Taentzer, G.; Crema, A.; Schmutzler, R.; Ermel, C.: Generating Domain-Specific Model Editors with Complex Editing Commands; in: Proc. Intl. Symp. Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007, Kassel; LNCS 5088, Springer; 2007

[29] Treude, Ch.; Berlik, S.; Wenzel, S.; Kelter, U.: Difference Computation of Large Models; p.295-304 in: Proc. Joint ESEC/FSE Conf., 2007, Dubrovnik; ACM; 2007

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

[31] Wierse, G: A Catalog of Modeling Patterns for the Unified Modeling Language; Technical Report, Philipps-Universität Marburg; 2010

[32] Xing, Z.; Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing; p.54-65 in: Proc. IEEE/ACM Intl. Conf. Automated Software Engineering 2005; IEEE Computer Society; 2005

[33] Xing, Z.; Stroulia, E.: Refactoring detection based on UMLDiff change-facts queries; p.263-274 in: Working Conf. Reverse Engineering 2006, Benevento; IEEE Computer Society; 2006

[34] Xing, Z.; Stroulia, E.: Differencing logical UML Models; Autom. Softw. Eng. 14:2, p.215-259; 2007

[35] Xtext Language Development Framework; http://www.xtext.org