User-Driven Adaptation of Model Differencing Results

Klaus Müller, Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen, Germany
http://www.se-rwth.de/

***Abstract*—In model-based software development, models are core development artifacts which are typically created and modified by multiple developers over a period of time. In order to be able to reason about the evolution of models, the computation of the differences between different versions of a model, called model differencing, is a crucial activity. However, in general a completely automatic approach to model differencing cannot infer the differences correctly in all cases. Errors in the reported model differences are particularly problematic, when the model differences are used in an automated process as a basis to perform other activities.**

In this paper, we propose an approach to model differencing in which users can integrate knowledge of how specific model elements changed from one model (version) to the other. By means of this, users can influence the reported model differences to avoid that wrong model differences are reported.

I. Introduction

In model-based software development, models are the central development artifacts which are usually transformed into software implementations [1]. However, due to changing requirements, the models often have to change too. Consequently, developers typically have to create and modify multiple versions of models over a period of time.

Due to this, dedicated support for model versioning and management is necessary to be able to cope with the evolution of models [2], [3]. One central functionality in the area of model versioning is model differencing, the computation of differences between two models. Model differencing is not only crucial for being able to reason about the evolution of models, but is also the basis for further model versioning functionalities such as model patching [4] or model merging [5], [6]. In this paper, the term model differencing refers to syntactic model differencing, intending to find structural changes in models. Other definitions to model differencing, e.g., semantic model differencing [7], [8] are not in the scope of this paper.

Despite the importance of model differencing, a completely automatic approach to model differencing cannot infer the differences correctly in all cases in general [9]. This can even hold for small changes like a renaming of a model element. For instance, a renaming of a class in a UML class diagram cannot only be regarded as a renaming but also as a deletion of the class with the original name and the addition of a class with the new name. The probability of such problems can be reduced, e.g., by integrating new concepts into model differencing tools. The previously mentioned problem regarding the renaming of a model element can, e.g., be addressed by checking linguistic databases such as WordNet [10] to take the semantics of strings into account when performing model differencing [9]. The problem that an automatic approach to model differencing cannot calculate the differences correctly in all cases especially holds when working with state-based model differencing approaches [11], [6]. Such approaches only utilize the state of the models to be compared to calculate the model differences. Alternatively, operation-based approaches [11], [6] could be used. In such approaches, the change operations that are performed by the modeler are basically ”recorded”. Although this can help to reduce the likelihood of wrong model differences, it does not guarantee that the reported model differences reflect the user expectations as well.

In principle, it is not necessarily critical if wrong model differences are reported, as long as users of model differencing tools are aware of this and, consequently, check the validity of the model differences. However, if the calculated model differences are used in an automated process as a basis to perform other activities, then wrong model differences are indeed problematic. In a cooperation project with an industrial partner, we are working on the generation of checklists, which inform developers about (potential) development steps that are necessary due to specific model changes [12]. For this purpose, at first the differences between two models are identified. If the model differencing activity reports wrong model differences, the resulting checklists might be unusable as parts of the checklists might have been created for wrong model differences. Although this is a special use case, the underlying problem is generally valid and also applies to other use cases in which the model differencing results are used to perform further tasks.

In this paper, we propose an approach to model differencing in which users can integrate knowledge of how specific model elements changed from one model (version) to the other. This knowledge is embodied by so-called user presettings and is stored in separate files. One user presetting instruction can, e.g., express that a specific model element should be regarded as renamed. For the sake of

simplicity, we limit examples in this paper to integrating knowledge of how model elements in UML class diagrams changed. We chose UML class diagrams as an example as UML class diagrams still represent the most frequently used modeling technique of the UML [13]. Despite this, the idea is applicable to other types of models too.

The paper is structured as follows: in Section II, we present examples for user presetting instructions before we give an overview of our approach in Section III. Afterwards, we outline the identification of relevant user presettings for a particular model comparison in Section IV before we briefly present related work in Section V. Finally, Section VI summarizes the paper.

II. Examples for user presetting instructions

The root cause for wrong model differences is that corresponding elements which are considered "the same" are not detected or that inappropriate elements are regarded as corresponding elements [9], [14]. As a consequence of this, user presetting instructions can be provided, which are capable to prevent the model matching engine from calculating wrong matchings. This aspect is explained in more detail in Section III.

In the cooperation project in which the idea of user presettings arose, there were mainly two situations which occurred in the model matching phase that led to wrong model differences:

- A model element was reported as deleted or added, although it was moved or renamed (or both).
- A model element was reported as renamed or moved (or both), although a model element was deleted and another model element was added.

To cope with these situations, the following user presetting instructions were derived:

- `added` "*modelElementRepresentation*": Indicates that the model element was added.
- `deleted` "*modelElementRepresentation*": Expresses that the corresponding model element was deleted.
- `moved` "*modelElementRepresentation*" `to` "*newModelElementLocation*": Indicates that the model element was moved.
- `renamed` "*modelElementRepresentation*" `to` "*newModelElementName*": Expresses that the respective model element was renamed.
- `moved and renamed` "*oldModelElementRepresentation*" `to` "*newModelElementRepresentation*": Expresses that the model element was moved and renamed.

Each affected model element has to be addressed through a unique string representation. Concrete examples for user presetting instructions are shown in Figure 1. The comments (indicated by the hash keys) above each user presetting concretize the meaning of each user presetting example.

```
1  # Addition of class Animal (in package de)
2  added "de.Animal";
3
4  # Deletion of association with name _Owner
5  # (of class Animal in package de)
6  deleted "de.Animal#_Owner";
7
8  # Moving of class Animal from
9  # package de into package de.shop
10 moved "de.Animal" to "de.shop";
11
12 # Renaming of attribute name from class
13 # TroubleCd (in package de) to newName
14 renamed "de.TroubleCd#name" to "newName";
15
16 # Renaming of class Animal (in package de)
17 # to OtherAnimal and moving of class into
18 # package de.se.
19 moved and renamed "de.Animal"
20   to "de.se.OtherAnimal";
```

Figure 1. User presetting instruction examples

Internally each user presetting instruction is basically treated like a mapping from a source model element to a target model element. Table I exemplifies this for the user presettings from Figure 1. The first two examples from Table I are special cases, as the added and deleted user presetting instructions express that no corresponding element exists in the other model. Due to this, the source element is `null` in the first case and the target model element is `null` in the second case. In all other cases each source model element can be mapped to a target model element.

Source element	Target element
1) `null`	`de.Animal`
2) `de.Animal#_Owner`	`null`
3) `de.Animal`	`de.shop.Animal`
4) `de.TroubleCd#name`	`de.TroubleCd#newName`
5) `de.Animal`	`de.shop.OtherAnimal`

Table I
Results of the model differencing component assessment

The decision to specify user presetting instructions as edit operations was motivated by discussions with developers in the aforementioned cooperation project. These discussions showed that most developers favored this notation over other notations such as stating user presetting instructions directly as mappings between a source and a target model element as indicated by Table I.

III. Overview

In order to perform model differencing including the possibility to process user presettings, we slightly extended the state-of-the-art model differencing tool EMF Compare [15]. Figure 2 gives an overview of the general workflow.

Like lots of model differencing tools EMF Compare performs model differencing in two phases: a matching and

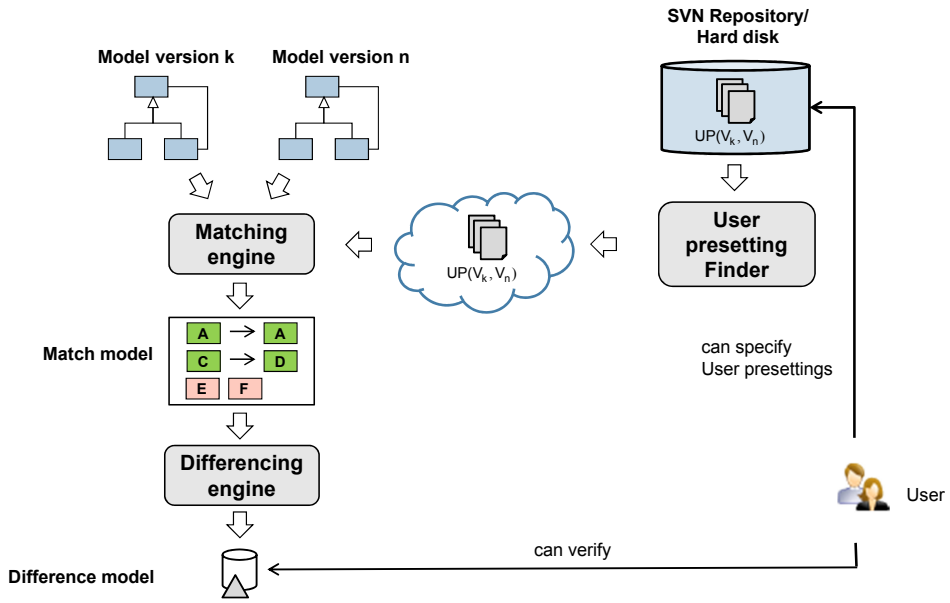

Figure 2. Overview of the integration of user presettings

a differencing phase [16], [3]. In the matching phase a match engine is responsible for finding the elements in both models that correspond to each other. The result of the matching phase is a match model, that reveals which model element in model version k is mapped to which model element in model version n. Moreover, the match model can contain unmatched elements, i.e., model elements of a model that could not be mapped to a model element of the other model.

The root cause for wrong model differences is that corresponding elements which are considered ”the same” are not detected or that inappropriate elements are regarded as corresponding elements [9], [14], as described in Section II. We address this problem by extending the matching phase by the processing of user presettings. This is outlined in the following.

Before the model matching engine is actually invoked, the user presetting file that is relevant for the particular model comparison is obtained. Section IV outlines this step. After that, it is at first validated whether the user presetting instructions only refer to model elements that really exist in the models. If a model element is specified that does not exist, the user is informed about that and the model matching is cancelled. In addition to that, it is validated that the user presetting instructions do not contradict each other. A contradiction occurs if a model element is reported as added and deleted at the same time or if two different user presetting instructions concern the same source or target model element. For instance, the instruction ’renamed ”`de.Y`” to ”`X`”’ contradicts the instruction ’moved ”`de.core.X`” to ”`de`”’ as both instructions would map different source elements to the same target model element `de.X`. The user is also notified about contradictory user presetting instructions and the model matching is cancelled in this case too.

After these validating steps, the actual model matching step is initiated. The model matching engine searches for matching model elements in the usual way, except it finds a model element for which a user presetting instruction exists. In this case, the model matching engine determines the model element to which the currently considered model element should be matched to according to the user presetting instruction. The resulting match element is then added to the match model, if the according user presetting instruction does not represent the addition or deletion of a model element. Otherwise an unmatched element is added to the match model as no matching element exists in this case.

The model differencing component finally builds the difference model based on the match model as usual. As the user presettings have been taken into account in the model matching step, the resulting difference model will not violate the user presetting instructions.

One aspect that has to be emphasized is that the only purpose of user presetting instructions is to influence the model matching for specific model elements. Consequently, user presetting instructions only have to be provided if the model matching engine would create a wrong match model otherwise. For all model elements that are not concerned by user presetting instructions, the model matching engine creates the match model as usual.

IV. User Presetting Retrieval

Before the actual model matching can be performed, the model matching engine has to be aware of the user presetting file that has to be taken into account in the model

matching phase. This user presetting file can either be passed directly to the model differencing tool or it is automatically checked if a proper user presetting file exists in the version control system. The benefit of storing user presetting files in a version control system[1] is that user presetting files can be obtained automatically each time the corresponding model versions are compared - no matter which developer invokes the model differencing tool. Lets assume a user presetting file has been created and commited when comparing a model with version i and with version k. The next time a developer performs the model differencing for the model versions i and k, the corresponding user presetting file can be used again without passing it explicitly to the model differencing tool. Thus the developer does not need to worry about creating or obtaining a user presetting on his own.

One prerequisite for such a behavior is that it is known which model versions a user presetting concerns. For this purpose, we established a naming convention for user presetting files which allows for inferring which model versions are concerned. The concrete naming scheme that is used is as follows: "up_<sourceRevision>_<targetRevision>". For instance, the user presetting file named "up_240_271" would concern a model with the revision 240 and the model with the revision 271. In addition to this naming convention, a further convention was applied to be able to derive to which concrete model file a user presetting refers to: if the model file is stored in the version control system under the path https://<parent_folder>/<model>, then the user presettings for this model are stored in the version control system in the folder https://<parent_folder>/up_<model>. These both conventions combined allow for identifying relevant user presetting files.

V. Related Work

The idea for user presetting instructions presented in this work was motivated by delta modeling approaches [17], [18], [19]. Delta modeling is a language-independent approach to model system variability. A set of deltas specifies modifications to a core system and each delta usually expresses that certain elements are added, removed, modified or replaced. Figure 3 shows how a delta that renames an element would typically look like in a delta modeling approach.

```
1 delta RenameElement {
2   modify class de.X {
3     Rename to Y;
4   }
5 }
```

Figure 3. Delta modeling example for the renaming of a model element

[1]Actually any storage location can be used to which every developer has access to.

The corresponding user presetting instruction 'renamed "de.X" to "Y"' is basically an abbreviated form of the delta operation. However, delta modeling is far more powerful than that and is typically used to create new versions of models - and not to capture differences in model versions. Furthermore, delta modeling cannot be used to adapt the results of a model differencing tool.

The adaptability of model differencing tools has been analyzed in [14] too. In [14] it is presented how model matching can be adapted to user preferences in the model differencing framework SiDiff. For instance, SiDiff provides the possibility to specify a custom algorithm for calculating the similarity of properties [20]. However, the focus in [14] is on configurable matching algorithms in general and not on guaranteeing that a particular model element is mapped to a particular other model element as in this work.

To the best of our knowledge, no other work focused on the user-driven adaptation of model differencing results by using a delta-like specification.

VI. Conclusion

In this paper, we have presented an approach that allows users to integrate knowledge of how specific model elements changed from one model (version) to the other - embodied in so-called user-presettings. This allows users to influence the results of a model differencing tool and is useful if reported model differences deviate from the expectations of a user.

This user-driven adaptation of model differencing results is particularly helpful when model differencing is used as an intermediary step and if the results are processed and used automatically to perform further steps. In such cases errors in the reported model differences can be especially problematic.

References

[1] R. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," in *Proc. Future of Software Engineering (FUSE'07)*, 2007, pp. 37–54.

[2] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayer, "Challenges in software evolution," in *Proc. International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005, pp. 13–22.

[3] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, "An introduction to model versioning," in *Proc. International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering (SFM'12)*, 2012, pp. 336–398.

[4] U. Kelter, T. Kehrer, and D. Koch, "Patchen von modellen," in *Software Engineering*, 2013, pp. 171–184.

[5] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A manifesto for model merging," in *Proc. International Workshop on Global Integrated Model Management (GaMMa'06)*, 2006, pp. 5–12.

[6] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002.

[7] S. Maoz, J. O. Ringert, and B. Rumpe, "A manifesto for semantic model differencing," in *Proc. International Workshop on Models and Evolution (ME'10)*, 2010, pp. 194–203.

[8] ——, "An interim summary on semantic model differencing," *Softwaretechnik-Trends*, vol. 32, no. 4, 2012.

[9] P. Pietsch, K. Müller, and B. Rumpe, "Model matching challenge: Benchmarks for ecore and bpmn diagrams," *Softwaretechnik-Trends*, vol. 33, no. 2, May 2013.

[10] WordNet Lexical Database (visited 02/2014). [Online]. Available: http://wordnet.princeton.edu/

[11] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt, "Operation-based model differencing meets state-based model comparison," *Softwaretechnik-Trends*, vol. 32, no. 4, 2012.

[12] K. Müller and B. Rumpe, "A Model-Based Approach to Impact Analysis Using Model Differencing," in *Proc. International Workshop on Software Quality and Maintainability (SQM'14)*, 2014.

[13] B. Rumpe, *Modellierung mit UML*, 2nd ed. Springer Berlin, September 2011.

[14] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt, "Adaptability of model comparison tools," in *Proc. International Conference on Automated Software Engineering (ASE'12)*, 2012, pp. 306–309.

[15] EMF Compare homepage (visited 02/2014). [Online]. Available: http://www.eclipse.org/emf/compare/

[16] M. Stephan and J. R. Cordy, "A Survey of Methods and Applications of Model Comparison," School of Computing, Queen's University, Tech. Rep. 2011-582 Rev. 3, June 2012.

[17] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented Programming of Software Product Lines," in *Proc. International conference on Software product lines (SPLC'10)*, 2010, pp. 77–91.

[18] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer, "Delta Modeling for Software Architectures," in *Tagungsband des Dagstuhl-Workshop: Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*, 2011.

[19] A. Haber, K. Hölldobler, C. Kolassa, M. Look, B. Rumpe, K. Müller, and I. Schaefer, "Engineering delta modeling languages," in *Proc. International Software Product Line Conference (SPLC'13)*, 2013, pp. 22–31.

[20] D. Kolovos, D. Di Ruscio, A. Pierantonio, and R. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Proc. ICSE Workshop on Comparison and Versioning of Software Models (CVSM'09)*, 2009, pp. 1–6.