Automatic Change Recommendation of Models and Meta Models Based on Change Histories

Stefan Kögel, Raffaela Groner, and Matthias Tichy
Institute of Software Engineering and Programming Languages
Ulm University
D-89069 Ulm
[stefan.koegel|raffaela.groner|matthias.tichy]@uni-ulm.de

ABSTRACT

Model-driven software engineering uses models and meta models as key artefacts in the software development process. Typically, changes in the models (or meta models) do not come in isolation but are part of more complex change sets where a single change depends on other changes, e.g., a component is added to an architectural model and thereafter ports and connectors connect this component to other components. Furthermore, these sets of related and depending changes are often recurring, e.g., always when a component is added to an architecture, it is highly likely that ports are added to that component, too. This is similar for changes in meta models. Our goal is to help engineers by (1) automatically identifying clusters of related changes on model histories and (2) recommending corresponding changes after the engineer performs a single change. In this position paper, we present an initial technique to achieve our goal. We evaluate our technique with models from the Eclipse GMF project and present our recommendations as well as the recommendation quality. Our evaluation found an average precision between 0.43 and 0.82 for our recommendations.

Keywords

Model-driven development; Change recommendation; Revision history mining

1. INTRODUCTION

As models are key artefacts in model-driven software engineering, software engineers typically spend much time creating and evolving models. Hence, they need good tool support to efficiently work with models.

There exist many simple or more complex tools in integrated development environments or code editors to improve the productivity of software engineers, for example, auto completion, quick fixes, refactorings, and templates for often used language constructs. These tools aim at improving development speed and quality. However, such tools are typically not available for changing models in model-driven software engineering (with the exception of autocompletion, e.g., in Xtext [6] based textual editors).

In our personal experience in modeling, we often had to perform repetitive and recurring changes when evolving the models. Furthermore, we sometimes forgot some individual changes in a model when performing complex changes.

Our hypothesis is that we can improve modeling speed and quality of model changes by recommending model changes to the engineer based on current changes and historical changes. For example, after adding a transition to a state machine, guards or actions are added to the transition afterwards.

Please note that the same argumentation holds for meta models as well. For example, often meta classes need to subclass a certain superclass. Therefore, after creating the meta class, it might be beneficial to recommend the addition of a generalization relationship to that meta class.

Recommender Systems aim at supporting users in making decisions. They recommend items of interest to users based on explicitly or implicitly expressed preferences [15]. An example of a recommender system in software engineering aims at proposing reuse possibilities in writing test cases [9]. Related to model-driven engineering, Brosch et al. presented a recommender addressing the conflict resolution in merging models [3]. However, there does not exist a recommender system to recommend modeling changes to the engineer as discussed before.

In this position paper, we propose a preliminary approach for generating live recommendations to engineers modifying models. Specifically, the approach recommends model changes to the engineer based on his currently performed changes where the to-be-recommended changes have been linked to the currently performed changes in historical change sets of the model.

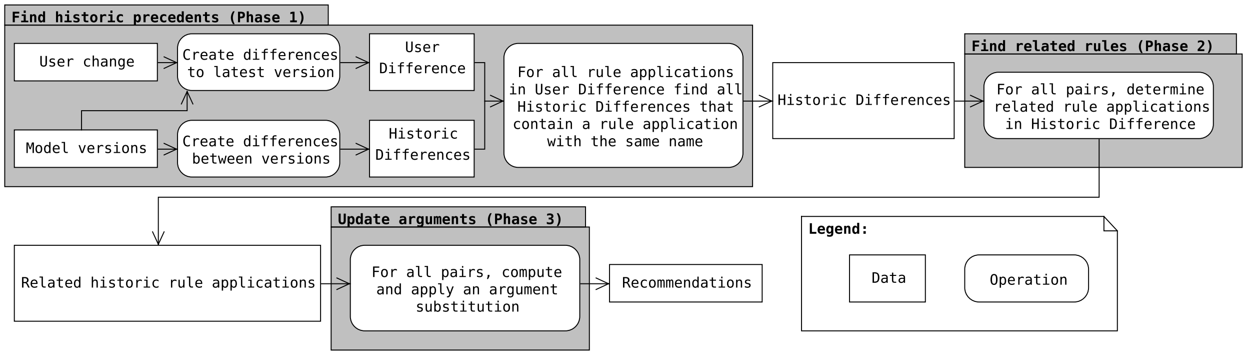
The aims of our approach are that it (G1) automatically produces recommendations, (G2) aggregates the recommendations in order to not overwhelm the user with too much information, and (G3) does not make too many wrong recommendations to prevent users from losing confidence.

As the approach requires a set of historical model changes, we mine model changes from version control systems and use SiLift [10, 11] to compute the individual model changes for each version. Our evaluation on several meta model histories show that we can reach medium to high precision.

We describe our approach for automatic change recommendation and several extensions in Section 2. In Section 3, we present an evaluation of our technique using several meta models from Eclipse Projects. Related work is discussed in Section 4. Section 5 concludes our paper and discusses future work.

2. TECHNIQUE

Our goal is to recommend further changes to a model based on current and historic changes. To achieve this goal we use the SiLift Tool [10] to compute the differences between historic versions of the same model. These differences are consistency preserving [11]. Figure 1 gives a high level overview of our technique. Rectangles represent data



Find historic precedents (Phase 1)
User change
Create differences to latest version
User Difference
Model versions
Create differences between versions
Historic Differences
For all rule applications in User Difference find all Historic Differences that contain a rule application with the same name
Historic Differences
Find related rules (Phase 2)
For all pairs, determine related rule applications in Historic Difference
Related historic rule applications
Update arguments (Phase 3)
For all pairs, compute and apply an argument substitution
Recommendations
Legend:
Data
Operation

Figure 1: Diagram of our technique

CreateEClassInEPackage	
↳ selectedEObject:	*gmfgraph*
↳ New:	*LayoutRef*

(a) Create a new EClass

AddEClassTgtEClass	
↳ selectedEObject:	*LayoutRef*
↳ NewTarget:	*Layout*

(b) Creates a super class relationship

CreateEAnnotationInEModelElement	
↳ selectedEObject:	*LayoutRef*
↳ New:	*LayoutRef/GenModel*

(c) Create an EAnnotation in an element

CreateEStringToStringMapEntryInEAnnotation	
↳ selectedEObject:	*LayoutRef/GenModel*
↳ New:	*LayoutRef/GenModel/details*

(d) Create string entry in EAnnotation

Figure 2: Excerpt from a difference with four Henshin rule applications and some of their arguments.

and rounded blocks represent operations on this data. We have divided our technique into three phases: (1) **finding historic precedents** for current user changes, (2) **finding related rules** for the historic precedents, and (3) **updating the arguments** of the historic precedents so that they match the current user changes. In the following, we will motivate and explain our technique using a running example. Note that the presented technique is a proof of concept and needs to be developed further.

2.1 Running Example

Figure 2 shows an excerpt from the difference between two historic model versions. The figure shows four Henshin [1] rule applictions, that consist of the rule names and a set of named arguments that contain, among other things, references to elements in models. In this example, the user has created an EClass identified by *LayoutRef* in the package *gmfgraph* (Figure 2a), set *Layout* as the superclass of *LayoutRef* (Figure 2b), created an EAnnotation in *LayoutRef* (Figure 2c), and created an entry in the EAnnotation (Figure 2d). Given this historic difference, suppose that a user adds a new EClass to the *gmfgraph* package. Our technique should then recommend that the new EClass should have a super class, an EAnnotation and an entry in this EAnnotation.

2.2 Basic Technique

In the following, M is a model and $M_i, 1 \leq i \leq n$ are its n different versions. The differences produced by SiLift $\delta_{i,i+1}$ contain partially ordered sets of Henshin [1] rule applications ($\text{rules}(\delta_{i,i+1})$).

We will now explain our basic technique for making recommendations based on historic differences between model versions.

Phase 1: Find historic precedents: When a user makes a change to the latest model version, we compute the difference $\delta_{current}$ between the latest model version and the current version. Then we look at every Henshin rule application in this difference and try to make a recommendation for it. Let $h_{current} \in \text{rules}(\delta_{current})$ be a current Henshin rule application, for example, the addition of a new EClass to the *gmfgraph* package.

First, we need to find historic rule applications that are related to our current rule application. Thus, we search for all historical differences $\Delta_{h_{current}}$ that contain a rule application with the same name as $h_{current}$:

$$\Delta_{h_{current}} = \{\delta_{i,i+1} | \\ h \in \text{rules}(\delta_{i,i+1}), \text{name}(h) = \text{name}(h_{current}), 1 \leq i \leq n\}$$

The technique will generate at least one recommendation for every historical difference that contains a rule application with the same name as $h_{current}$.

Phase 2: Find related rules: Given a rule application $h_{hist} \in \delta, \delta \in \Delta_{h_{current}}$ with the same name as $h_{current}$, we now have to determine which other rule applications in the same difference are related to it. To do this, we search for all rule applications that have an argument in common with h_{hist}, i.e., they both have a reference to the same model element. Note that the steps in **Phase 2** are repeated for every historical rule application with the same name as $h_{current}$.

$$H_{h_{hist}} = \{h | h \in \delta, \delta \in \Delta_{h_{current}}, \text{args}(h) \cap \text{args}(h_{hist}) \neq \emptyset\}$$

In our running example, we determine that the rule applications in Figures 2a, 2b, and 2c are related because they have *LayoutRef* in common, which is not the case for the rule application in Figure 2d.

Phase 3: Update arguments: Given all the related rule applications $H_{h_{hist}}$ from a historic difference δ, we need to update their arguments so as to fit to the current user

CreateEClassInEPackage
↳ selectedEObject: **gmfgraph**
↳ New: **RealFigure**

(a) Create a new EClass

AddEClassTgtEClass
↳ selectedEObject: **RealFigure**
↳ NewTarget: *Layout*

(b) Creates a super class relationship

CreateEAnnotationInEModelElement
↳ selectedEObject: **RealFigure**
↳ New: *LayoutRef/GenModel*

(c) Create an EAnnotation in an element

Figure 3: Figure 3a is a rule application describing a user change. Figures 3b and 3c are recommendations based on the user change and the history in Figure 2.

change $h_{current}$. We use the current rule application $h_{current}$ and its historic counterpart $h_{hist} \in \delta$ with the same name as a guide to compute a substitution *subst* between their arguments.

$$\mathit{subst}(\text{value}(p_{hist})) = \text{value}(p_{curr}) \text{ where}$$
$$p_{hist} \in \text{args}(h_{hist}), p_{curr} \in \text{args}(h_{current}),$$
$$\text{name}(p_{hist}) = \text{name}(p_{curr})$$

Applying this substitution to all rule applications in $H_{h_{hist}}$ results in a set of updated Henshin rule applications that can be applied to the current model (assuming there are no constraint violations).

$$\mathit{Recommend}(h_{current}, h_{hist}, \delta) = \{\mathit{subst}(h) | h \in H_{h_{hist}}\}$$

This new set of Henshin rule applications is one of our recommendations. Note that our technique makes a recommendation for every historic rule application h_{hist} with the same name as the current rule application $h_{current}$.

In our running example, Figure 3a shows the rule application *CreateEClassInEPackage* with an argument value of **RealFigure**. When we use the rule application as a user change and Figure 2 as historic rule applications, our technique would find the related historic rule applications in Figures 2a, 2b, and 2c, because they have *LayoutRef* in common. Note that the rule in Figure 2d has no arguments in common with the one in Figure 2a and, hence, would not be found. The corresponding substitution would be:

$$\mathit{subst} = [\mathit{gmfgraph} \rightarrow \textbf{gmfgraph}, \mathit{LayoutRef} \rightarrow \textbf{RealFigure}]$$

This would lead to the recommendations shown in Figures 3b and 3c. Note that both recommendations now reference **RealFigure** instead of *LayoutRef*, while they have kept their references to *Layout* and *LayoutRef/GenModel*.

There are two problems with these recommendations: (1) There is no recommendation for Figure 2d, even though it is related to the recommendation in Figure 3c via its *LayoutRef/GenModel* argument. (2) The historic references *Layout* and *LayoutRef/GenModel* are carried over to the recommendation without changes. It is possible that these historic references lead to correct recommendations, for example, if most newly created EClasses are sub classes of *Layout*. But it would still be better if we could control their inclusion via a tuning parameter, e.g., by keeping or replacing them with a placeholder value. In the next section, we will present extensions to our technique that deal with these problems.

2.3 Extensions to our Technique

We have implemented extensions to our technique to improve the quality of our recommendations with respect to our quality goals (G1-G3).

2.3.1 Intersection of recommendations

So far, we have only described how recommendations are generated. If there are multiple applicable recommendations (from multiple historical rule applications of the same rule), we have to aggregate them before presenting them, because showing too many recommendations will frustrate users. A simple solution would be to count recommendations that contain the same rules and to rank them accordingly. This is similar to frequency based completion in [4].

In this extension, we propose a different solution. Because recommendations are sets of rule applications, we can compute the intersection between all recommendations. To do this, we define that two rule applications are equal if their names are equal [1]. In this way, we merge several recommendations, originating from the same current user change, into a single one (G2). Furthermore, by reducing the amount of rule applications in a recommendation, we reduce the chance that a recommendation will recommend changes that are not intended by the user (G3). This extension takes place after **Phase 3**.

As an example, suppose we have three recommendations consisting of rule applications (ignoring arguments):
$R1 = \{\mathit{addClass}, \mathit{addEdge}, \mathit{deleteClass}\}$,
$R2 = \{\mathit{addClass}, \mathit{addAnnotation}, \mathit{addEdge}\}$,
and $R3 = \{\mathit{addClass}, \mathit{deleteEdge}, \mathit{addEdge}\}$
then the intersection of recommendations $R1 \cap R2 \cap R3$ would be $\{\mathit{addClass}, \mathit{addEdge}\}$. Because *addClass* and *addEdge* have always occurred simultaneously in all recommendations, we suspect that they should occur together again. *deleteClass*, *deleteEdge*, and *addAnnotation* would not be presented to the user, because we can not decide which of these rule applications is the most likely to be intended and they have never occurred simultaneously in the recommendations.

A drawback of this extension is that correct recommendations may be discarded. Another problem is that the intersection of all recommendations may be empty, but this is easily detected and we can fall back on a different method of reducing the amount of recommendations.

2.3.2 Free Variables

In our running example, we cannot predict to which element the argument *LayoutRef/GenModel* in Figure 2c/3c should refer. So we extend our substitution *subst* from **Phase 3** to substitute the historic element with a place holder, a free variable. Replacing all references to unpredictable elements in the arguments leads to the recommendation in Figure 4. Note that the rule applications in Figures 4b and 4c now refer to free variables `Free1` and `Free3` instead of keeping their references from Figure 2. Also note, that rule applications in Figures 4c and 4d are now related

[1]We have not yet extended this equality to consider arguments.

by the free variable `Free2`, instead of by *LayoutRef/GenModel* as in Figures 2c and 2d.

The introduction of free variables removes all references that were only part of the historic rule applications, while preserving the relations between the rule applications. This makes recommendations more abstract (G3), but also requires users to manually fill in values for the free variables.

2.3.3 Indirect Relations

In our running example (Figures 2 and 3), we have, so far, only recommended two rule applications (Figures 3b and 3c). Figure 2d is not part of the recommendation, because it does not share an argument with the rule application in Figure 2a.

In order to find more related historical rule applications for recommendations, we also look at indirectly related rules. Two rules are related if they have a model element in their arguments in common. Two rules are indirectly related, if they have no elements in common, but there is a third rule that has arguments in common with both. For example Figure 2a and Figure 2c have the element *LayoutRef* in common, while Figure 2c and Figure 2d have the element *LayoutRef/GenModel* in common.

We implement this in our technique by iterating the search for related rule applications in **Phase 2**. After finding all rule applications that are directly related to h_{hist}, we repeat the search for all rule applications that have been found.

$$H_{h_{hist},0} = \{h | h \in \delta, \delta \in \Delta_{h_{current}}, \text{args}(h) \cap \text{args}(h_{hist}) \neq \emptyset\}$$

$$H_{h_{hist},p} = \{h | h \in \delta, \delta \in \Delta_{h_{current}}, h_{ind} \in H_{h_{hist},p-1}, \text{args}(h) \cap \text{args}(h_{ind}) \neq \emptyset\}$$

This search can be iterated for a user defined amount of time or until a fixed point is reached and no more indirectly related rule applications can be found.

Indirect relations introduce many rule applications whose arguments will not be changed by the argument substitution *subst*, because the indirectly related rules have no arguments in common with h_{hist}. This will lead to the recommendation of rule applications with many historical arguments. This can be prevented by using free variables.

This extension can also lead to a very high number of predicted rule applications that needs to be reduced again. We found that filtering out all sets of related rule applications for which $|H_{h_{hist},p}| > x$ leads to a better precision.

Using indirect relations allows us to make more complex recommendations, by including more rule applications that are not directly related to the user's changes (G1). This could lead to too specific or complex recommendations that do not fit the user's intentions. We can control this, by adding a parameter that limits the amount of iterations in the search for indirectly related elements (G2). A value of 0 for this parameter turns this extension off, while a value of 1 only allows indirections through one variable (as depicted in the example above), and a value of infinity leads to the search for a fixed point.

3. EVALUATION

In this section, we will describe the data set we used in our evaluation, how we evaluated our technique, and what results we achieved.

CreateEClassInEPackage
↳ selectedEObject: **gmfgraph**
↳ New: **RealFigure**

(a) User added EClass identified by *RealFigure*

AddEClassTgtEClass
↳ selectedEObject: **RealFigure**
↳ NewTarget: `Free1`

(b) Recommended rule application

CreateEAnnotationInEModelElement
↳ selectedEObject: **RealFigure**
↳ New: `Free2`

(c) Recommended rule application

CreateEStringToStringMapEntryInEAnnotation
↳ selectedEObject: `Free2`
↳ New: `Free3`

(d) Recommended rule application

Figure 4: Rule application that describes a user change (4a) and recommended rule applications (4b, 4c, 4d). Based on the historic rule applications in Figure 2

Herrmannsdörfer et al. [8] and Langer et al. [12] have analysed the versions of three different meta models from the Eclipse GMF Project. Kehrer et al. [11] have already applied SiLift to this data set and shown that it can compute correct and complete differences between all versions. The data set consists of the following meta models: (1) `gmfgen` with 110 model versions from 1.139 to 1.248, (2) `gmfgraph` with 10 model versions from 1.23 to 1.33, and (3) `mappings` with 15 model versions from 1.43 to 1.58.

While we use meta model histories in our evaluation since they are readily available, our approach itself is applicable to models as well which we will evaluate in the future.

First, we created asymmetric differences between all versions with SiLift, specifically using its UUID matcher and its atomic rule set. We found the following numbers of Henshin rules per meta model: (1) `gmfgen` 1067, (2) `gmfgraph` 163, and (3) `mappings` 149. Then we extracted the names of the Henshin rules and their arguments from the differences. For every difference $\delta_{i,i+1}$ we used all previous differences $\delta_{j,j+1}, j < i$ as historic differences and the rule applications in $\delta_{i,i+1}$ as the current user changes. That means the changes we wanted to recommend were not part of the historic changes.

We used the current difference $\delta_{i,i+1}$ to validate our recommendations. For every recommended rule application, we tried to find a rule application with the same name and arguments in the current difference. Free variables in the arguments were always counted as the same. A correctly recommended rule application, with correct arguments (or free variables), was counted as one true positive (TP), else it was counted as false positive (FP). Note that the usage of free variables increases the true positive count, but that users also need to do more manual work, which we do not measure here.

Because we do not try to predict the absence of certain rule applications, we can not measure true or false negatives.

We have summarised our results in Table 1 The table shows the number of true (TP) and false positives (FP) per

Name	TP	FP	Precision
gmfgen	554	217	0.72
gmfgraph	25	33	0.43
mappings	36	8	0.82
Total	615	258	-
Average	205	86	-

Table 1: Recommendation results for different models

model. The *Precision* metric is computed by the formula $Precision = \frac{\text{TP}}{\text{TP+FP}}$ and can be interpreted as the percentage of correctly recommended rules. The results were obtained using all extensions that were discussed in Section 2.3: intersection of recommendations, free variables, and indirect relations. The indirect relations were iterated two times and sets of related historic rule applications with more than five rules were ignored.

Internal validity: Our Evaluation only included three meta models from Eclipse Projects. **External validity:** We have not shown that these models are representative for all meta models or instances of meta models. Furthermore, the batch evaluation preformed in this paper is no substitute for an evaluation with real users as Turpin and Hersh [16] have shown.

4. RELATED WORK

There are many studies and tools about evolving models and meta models, but only a few of them look at model histories and try to automatically recommend changes based on user changes.

Herrmannsdörfer et al. [8] have analysed meta models in the Eclipse GMF Project and developed a set of small change operations that can be used in aggregate to describe all changes to the meta models. Langer et al. [12] developed this idea further by identifying complex change operations that consist of smaller ones. This allows the analysis of a model's evolution on a more abstract level. They have shown that their complex change operations can describe all model changes in the Eclipse GMF Project.

Kehrer et al. [11] have developed a tool that can automatically generate consistency-preserving edit scripts that describe the difference between two versions of the same model. They have implemented the small and complex change operations from [8] and [12] in their tool and evaluated it on the meta models from the Eclipse GMF Project. Their tool could correctly produce edit scripts for all model versions.

In this paper, we have used some of the results from [8], [12], and [11]. Mainly in the form of SiLift for generating our differences and in the form of the Eclipse GMF meta models for our evaluation.

Getir et al. [7] proposed a framework for model co-evolution. Their framework uses model histories, SiLift and Henshin rules to produce suggestions for coupled simultaneous model changes in related instance models. The framework requires manual intervention from developers with domain knowledge. Developers have to add traces between related elements in different models, in order to enable the framework to recognize the relations. The models' version histories and the traces are then used to propose further changes to one model based on user edits in another model. A correlation analysis between change operations in the models' histories is used to predict related change operations. The main difference to our work is that we analyse the arguments of rule application in order to identify related operations. Furthermore, our technique is able to recommend some of the rule application arguments.

Cicchetti et al. [5] present an approach for updating instance models whose meta models have changed. They compute a transformation from the changes in the meta model that can be applied to the instance models so that they conform to the new version of the meta model. This approach does not consider historic model changes and does not try to recommend further changes in the same model based on user changes.

LASE [13] is a tool that can create abstract edit scripts from several similar source code changes made by a user. It matches the change operations in the AST from similar changes, keeping common change operations and abstracting over similar change operations that differ only in variable or method names. The tool can then search for further source code regions that match abstract edit scripts and apply them automatically. This is similar to our technique, although we use historic changes to generate recommendations for a single change made by a user.

Breckel [2] evaluated an approach for automatic error detections in source code. Their approach finds similar but not identical code fragments in a source code file and large code bases. Frequent differences in these code fragments point to bugs in the source code file. This approach can detect typing and more complex errors, is programming language independent, and does not require domain knowledge. This is similar to our technique which automatically compares different versions of models without using domain knowledge about these models.

Bruch et al. [4] developed several systems that incorporate code from repositories to improve the auto completion features of Eclipse. Their systems rank auto completions by finding similar code in the repositories that share variables of the same types. They show that their systems outperform Eclipse code completion. The systems are used to rank auto completions recommended by Eclipse, while our technique generates recommendations based on the change histories of models.

Muşlu et al. [14] developed a technique to improve Eclipse Quick Fix. Their technique automatically counts how many errors are solved or introduced by a quick fix and displays this information to the developer. They improve the source code Editing capabilities of Eclipse by incorporating additional information from the compiler. This technique helps developers to decide between different quick fixes that are already implemented in Eclipse, but does not generate completely new recommendations on its own.

5. CONCLUSIONS AND FUTURE WORK

We have presented our technique for recommending changes based on historical changes. The precision of our recommendations in the evaluation was between 0.43 and 0.82. This shows that our technique could possibly be used to automatically recommend model changes based on user changes and previous model versions.

We plan to improve our technique by addressing the following topics:

- The current aggregation of multiple recommendations via intersections needs to be compared to other techniques, for example from [4], for prioritizing recommendations.

- The recommendations are based on single user changes. Future work should take multiple user changes into consideration.
- Our evaluation only included meta models. We plan to also evaluate our technique for types of instance model for which SiLift can generate differences.
- Our technique needs to be integrated into an intuitive user interface and evaluated by users, because batch evaluations alone are not sufficient [16].
- An extension of our technique that exploits model constraints could filter out recommendations that violate model constraints. It could also be possible to emphasize recommendations that fix constraint violations, which would be similar to Eclipse's Quick Fix recommendations. Muşlu et al. [14] have already done similar work for Eclipse Quick Fixes.
- We want to extend our technique to multiple meta and instance models that are evolving simultaneously. For this, we suspect that it is possible to find relations between changes that are made simultaneously to different models. For example, if two elements are added simultaneously to two different models, we could deduce a relation between them and then try to apply our technique.

6. ACKNOWLEDGEMENTS

This work was funded by the German Research Foundation (DFG) as part of the DFG Priority Programme 1593 (SPP1593).

7. REFERENCES

[1] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[2] A. Breckel. Error mining: bug detection through comparison with large code databases. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 175–178. IEEE Press, 2012.

[3] P. Brosch, M. Seidl, and G. Kappel. A recommender for conflict resolution support in optimistic model versioning. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA*, pages 43–50, 2010.

[4] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[5] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*, pages 222–231. IEEE, 2008.

[6] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[7] S. Getir, M. Rindt, and T. Kehrer. A generic framework for analyzing model co-evolution. In *Model Evolution, International Conference on Model Driven Engineering Languages and Systems*, 2014.

[8] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: The history of gmf. In *International Conference on Software Language Engineering*, pages 3–22. Springer, 2009.

[9] W. Janjic and C. Atkinson. Utilizing software reuse experience for automated test recommendation. In *8th Int. Workshop on Automation of Software Test*, pages 100–106, 2013.

[10] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 638–641. IEEE, 2012.

[11] T. Kehrer, U. Kelter, and G. Taentzer. Consistency-preserving edit scripts in model versioning. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 191–201. IEEE, 2013.

[12] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[13] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. IEEE Press, 2013.

[14] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices*, 47(10):669–682, 2012.

[15] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer, 2014.

[16] A. H. Turpin and W. Hersh. Why batch and user evaluations do not give the same results. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 225–231. ACM, 2001.