Recommender System for Model Driven Software Development

Stefan Kögel*
Institute of Software Engineering and Programming Languages, Ulm University
Ulm, Germany
stefan.koegel@uni-ulm.de

ABSTRACT

Models are key artifacts in model driven software engineering, similar to source code in traditional software engineering. Integrated development environments help users while writing source code, e.g. with typed auto completions, quick fixes, or automatic refactorings. Similar integrated features are rare for modeling IDEs. The above source code IDE features can be seen as a recommender system.

A recommender system for model driven software engineering can combine data from different sources in order to infer a list of relevant and actionable model changes in real time. These recommendations can speed up working on models by automating repetitive tasks and preventing errors when the changes are atypical for the changed models.

Recommendations can be based on common model transformations that are taken from the literature or learned from models in version control systems. Further information can be taken from instance- to meta-model relationships, modeling related artifacts (e.g. correctness constraints), and versions histories of models under version control.

We created a prototype recommender that analyses the change history of a single model. We computed its accuracy via cross-validation and found that it was between 0.43 and 0.82 for models from an open source project.

In order to have a bigger data set for the evaluation and the learning of model transformation, we also mined repositories from Eclipse projects for Ecore meta models and their versions. We found 4374 meta models with 17249 versions. 244 of these meta models were changed at least ten times and are candidates for learning common model transformations.

We plan to evaluate our recommender system in two ways: (1) In off-line evaluations with data sets of models from the literature, created by us, or taken from industry partners. (2) In on-line user studies with participants from academia and industry, performed as case studies and controlled experiments.

CCS CONCEPTS

• **Software and its engineering → System modeling languages**;

KEYWORDS

Model Driven Software Engineering; Recommender System; Data Mining; Machine Learning; Heuristic Search Algorithms

ACM Reference Format:
Stefan Kögel. 2017. Recommender System for Model Driven Software Development. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 4 pages.
https://doi.org/10.1145/3106237.3119874

1 INTRODUCTION

Models are key artifacts in model driven software engineering. They offer a higher level of abstraction than source code, speed up development through the generation of code from them, and simplify the development of domain specific languages [3].

There are many tools that help software engineers during the development of models, but most of these are analysis tools that are used after changing models. An example is the recommender for conflict resolution in merging models from Brosch et al. [4].

A recommender system uses data about previous actions or expert knowledge to generate lists of possible future user actions based on current user actions [15]. Recommender systems are often used to propose music, films, or products to users based on their previous behavior.

Simple recommender systems for source code are available in almost every development environment, e.g. simple auto completions of names, templates for method stubs or loops, and refactoring tools that automate common operations.

We present a vision for a recommender system that can be used during modeling, that supports users by speeding up their work and notifying them about possible errors.

In our vision, when a user changes a model, the recommender system will show a list of further model changes based on the current ones. These lists of model changes can be based on three data sources:
(1) past changes to the currently changed model as recorded in a version control system [11].
(2) additional user defined meta data or artifacts, i.e. documentation, explicit traces between model elements [6], and consistency constraints (e.g. OCL) [21]
(3) common changes to models of a similar type that can be gathered from literature [2] or mined via machine learning [20] from data sets of models under version control.

Models are much more abstract than source code, but there is still repetition during modeling that can be automated. In the following, we present possible use cases:

*Advisor: Matthias Tichy; EMail: matthias.tichy@uni-ulm.de

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3119874

- In a class diagram, most classes will have a super class or other associated classes. When a user adds a new class, a recommender system can give the user a hint to click on another class. If the user accepts the hint, a new reference link to the clicked class can be inserted automatically. The type of the reference (composition, inheritance, etc.) can be based on the most common type of reference in the diagram.
- Some projects require that every element in a certain model has a documentation attribute. A recommender can infer this requirement automatically from this model and highlight elements that do not yet have such an attribute.
- A user may start refactoring a model manually, for example by moving all common attributes in a set of subclasses into their parent class [1]. A recommender can then check if the manual changes match an expert defined model transformation and automatically recommend the remaining steps of the refactoring.
- The classes in a class diagram can also appear in a sequence diagram. If a class is renamed or if its methods change, then the sequence diagram might need to be updated so that the two models remain consistent with each other. A recommender system that is integrated into an IDE can use inter-model traces between related elements to notify users of such possible inconsistencies [6].

From the above, we infer four aims for our recommender system:
Aim 1: Recommendations are relevant to the currently edited model and actionable for the software engineer.
Aim 2: The recommender has information about the model's history and other related models in order to make recommendations based on past changes and related elements in other models.
Aim 3: The recommender will be be evaluated off-line with data sets and on-line in case studies and controlled experiments.
Aim 4: The user interface is good enough so that users don't get frustrated by usability problems and stop using the recommender.

We give an overview of related work in Section 2. Section 3 presents our approach for developing the recommender system. This section also describes our results achieved so far and outlines our plan for evaluation. Section 4 contains a short summary of this paper.

2 RELATED WORK

Ricci et al. [15] describe fundamental recommender techniques (**Aim 1**), evaluation techniques (**Aim 3**), and usability issues (**Aim 4**). They also discuss how user generated content can be incorporated into recommender systems, which is relevant to our approach of learning recommendations from how users change models.

Robillard et al. [16] focus on recommender systems for software engineering. For these they identify the challenge of automatically interpreting technical data stored in software repositories (**Aim 2**).

For **Aim 1** we need to generate and rank recommendations:

Sen et al. [18] present a system that completes an instance model so that it becomes valid to its meta model. While their system does not take the instance model's history or related models into account, it could be used for generating recommendations for incomplete models. (They also implemented a user interface for their system, which is relevant for **Aim 4**.)

Bruch et al. [5] developed and evaluated several techniques to improve the auto completion for source code in Eclipse. They rank auto completions by counting how often they match code fragments from the software engineers source code repository. Their techniques can also be applied to ranking recommendations for models.

For **Aim 2** we need to analyze and represent the changes to models over time:

Herrmannsdörfer et al. [7, 8] have analyzed Ecore meta models [19] from the Eclipse Graphical Modeling Framework[1]. They inferred a set of atmoic change operations that can be combined to describe all changes to these models. Langer et al. [13] developed this idea further by identifying complex change operations that consist of the atmoic change operations. These complex change operations can be used to describe changes to a model on a more abstract level in our recommender.

Kehrer et al. [10] developed the tool SiLift. This tool automatically generates consistency-preserving model transformations that describe the difference between two versions of the same model. These models can be instance or meta models, as long as they are based on EMF and their meta models are available. The transformations are made up of the above atomic and complex change operations, represented as Henshin [1] rule applications. We use this tool in our work to analyze changes between model version.

The second part of **Aim 2** encompasses the analysis of inter model relations:

Getir et al. [6] proposed a framework for model co-evolution, where users generate traces between related elements in different models. This approach requires additional information from the software engineer. When this data is available, it can be used to improve recommendations.

Ricci et al. [15] give two main approaches for evaluating recommender systems that are relevant for **Aim 3**:
(1) In an off-line evaluation, an appropriate data set is split into a training and evaluation set and used for generating recommendations and validating these. But this is not sufficient on its own, because no human subjects are involved.
(2) An on-line evaluation can be done as a controlled experiment (Wohlin et al. [22]) or as a case study (Runeson et al. [17]). Both approaches require human subjects.

Because there are few recommender systems for modeling, we looked at usability studies for other types of recommender systems in order to achieve our **Aim 4**:

Zins et al. [23] evaluated a travel recommendation system using a combination of objective and subjective metrics. They found that there are three factors for user satisfaction: ease-of-use/learnability, effectiveness/outcome, and reliability. These factors are also relevant for a software development related recommender system.

3 PROPOSED APPROACH

In order to implement our proposed recommender and achieve the aims, we propose the following approach.

Figure 1 shows an outline of the artifacts required to develop our recommender system. The plan consists of four phases: data gathering, analysis (**Aim 1** and **2**), evaluation (**Aim 3** and **4**), and

[1] https://www.eclipse.org/modeling/gmp/

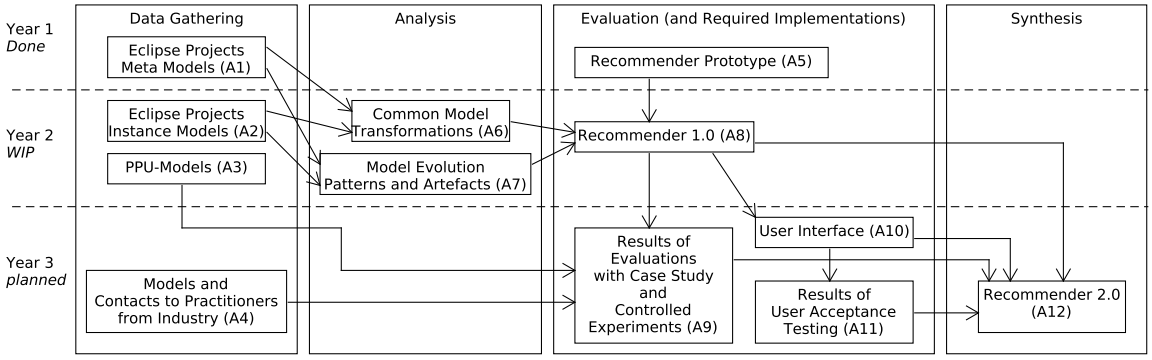

Figure 1: Required artifacts for finishing the recommender. Sorted by project phase and estimated finishing time. Dependencies are depicted as arrows.

synthesis. Some of these phases will be worked on in parallel. The artifacts are ordered according to their dependencies. Along the x-axis is a rough time plan indicating when the artifacts will be done, or which artifacts (**A[1-12]**) have already been finished.

Data gathering: Recommendations cannot be made in a vacuum. Additional context information about a modified model needs to be gathered in order to make relevant recommendations. We see two main approaches here:
(1) Similar models and their change histories can be analyzed for common changes and refactorings. These can be precomputed and integrated into the recommender.
(2) The history or meta information of the model that is currently worked on can be analyzed to generate specialized recommendations for this exact model. This needs to be computed on the user side.

In respect to point (1) above, we mined open source git repositories hosted on Eclipse projects for Ecore meta models [12] (**A1**). We found 4374 Ecore meta models with 17249 versions from Eclipse projects. 244 of these meta models were changed at least ten times. These changes were distributed over the whole lifetimes of the models and give insights into how meta models are changed by users.

We plan to use this data set to learn common model transformations for Ecore meta models [9, 20] and to use it for off-line evaluations of our recommender system. Common model transformations can be used as blueprints for recommendations by matching them to changes made by a user. When a common transformation becomes a good enough match for a sequence of user changes, the remaining changes in the transformation can be recommended to the user. This is similar to a live application of model repair [21]. Note that we will also have to investigate what *good enough* means.

The Ecore meta model data set can also be used as a starting point for identifying instances of these meta models in Eclipse projects (**A2**). These instance models will enable us to learn additional common model transformations for other types of models, increasing the generality of our recommender.

Our work is part of a joint project that uses a common demonstrator called the Pick-and-Place-Unit [14] (PPU). The PPU is a bench-scale manufacturing system that is used as an open case study for studying evolution of automation systems. We plan to create further models based on the PPU in a case study of our own in order to further evaluate our recommender system (**A3**).

For our final data set we plan to use our contacts to industry to get access to models and practitioners from an industrial setting (**A4**) in order to further evaluate our recommender system.

Analysis: The quality of recommendations depends in part on the availability of information about how certain types of models are changed, or how they evolve.

Based on our data sets of models and their meta data (from **A1** and **A2**) we plan to apply machine learning techniques and heuristic search algorithms similar to [20] in order to learn common model transformations (**A6**). For this, we currently evaluate an approach where model transformations are represented as sets of atmoic change operations[10] and we use a genetic algorithm to find model transformations that appear often in the data set from **A1**.

Another possible approach is applying association rule learning or random decision forests to our data set of model versions **A1**. The resulting rules or decision trees can then be used to generate recommendations.

Note that the above approaches will not necessarily generalize between different types of models.

To get a better insight into how models and their related artifacts (generated code, OCL constraints, instances of meta models, etc.) evolve over time we will also perform further studies of the artifacts contained in the repositories identified in **A1**. We will use this result to decide which artifacts are most helpful for generating and improving recommendations (**A7**).

Evaluation: We have created a prototype recommender [11] (**A5**) that represents recommendations as Henshin rule applications [1]. This recommender computes the atmoic change operations between all consecutive versions of a model including the most recent change by a user. It then searches for matches of the user change in the historic changes. If there is a match, it is extended by related atmoic change operations from the historic changes. Here model transformations are related if they simultaneously modified the same elements in the model. For every related model transformation, one recommendation is generated. Finally, the recommender aggregates all recommendations to a single one by computing the

atmoic change operations that are common to all generated recommendations.

We evaluated this prototype with a data set from Hermannsdörfer et al. [7, 8]. Our main results were that recommendations can be generated based on a model's change history with a precision between 0.43 and 0.82, but further work is required in order to rank the recommendations based on relevance and to aggregate many similar recommendations into a single actionable one. Note that the above precision values are from a simple prototype and that we expect to further improve them by adding more complex techniques.

Another result was that it is possible to reuse historical changes as recommendations by adapting them to the current changes made by a user. Although such an adaptation cannot always be done automatically and some user input is required.

For example, our recommender infers for our test data set that most of the time when a new node is added to an Ecore meta model it should also have a reference to a super class. However it is difficult in the general case to determine to which class this reference should point.

There are techniques for source code recommenders that improve the relevance of their recommendations, e.g. [5].We plan to adapt these in order to rank and aggregate model recommendations. Furthermore, we can combine them with our results from **A6** and **A7**, leading to an improved version of our prototype recommender system (**A8**).

This new recommender system will be evaluated with data from our research partners (**A3**) and from industry contacts (**A4**). We will then publish the results in order to gain further feedback (**A9**).

We plan to integrate our recommender system into a common modeling IDE like Eclipse (**A10**), in order to evaluate it with users in a case study (**A11**).

Synthesis: Our final step will be to integrate the results from the evaluation (**A9** and **A11**) and to further develop the recommender system (**A8**) and its user interface (**A10**) into a final version (**A11**).

4 SUMMARY

We have presented a vision for a recommender system for model driven software development. The recommender system combines state of the art approaches from model driven software development (e.g. model repair, model refactorings, learning of model transformations) with approaches from source code recommender systems (e.g. type based auto completions, refactorings, quick fixes). It can be used to speed up modeling by automating repetitive tasks and to ensure correctness by warning users when they violate consistency constraints (both intra- and inter-model) or when they make atypical changes to a model.

In order to develop and evaluate the recommender, high quality data sets of models, their evolutions, and their related artifacts will be created and made available to other researchers. Common model transformations and evolution patterns will be learned from these data sets and will also be made available.

ACKNOWLEDGMENTS

This work was partially supported by the DFG (German Research Foundation) (grant numbers TI 803/2-2 and TI 803/4-1).

REFERENCES

[1] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: advanced concepts and tools for in-place EMF model transformations. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 121–135.

[2] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. 2006. EMF Model Refactoring based on Graph Transformation Concepts. *ECEASST* 3 (2006). http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/34

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering* 1, 1 (2012), 1–182.

[4] Petra Brosch, Martina Seidl, and Gerti Kappel. 2010. 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*. 43–50. DOI: http://dx.doi.org/10.1145/1869542.1869549

[5] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. 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*. ACM, 213–222.

[6] Sinem Getir, Michaela Rindt, and Timo Kehrer. 2014. A generic framework for analyzing model co-evolution. In *Model Evolution, International Conference on Model Driven Engineering Languages and Systems*.

[7] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. 2009. COPE-automating coupled evolution of metamodels and models. In *European Conference on Object-Oriented Programming*. Springer, 52–76.

[8] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. 2009. Language evolution in practice: The history of GMF. In *International Conference on Software Language Engineering*. Springer, 3–22.

[9] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. 2012. Model Transformation By-Example: A Survey of the First Wave. In *Conceptual Modelling and Its Theoretical Foundations - Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday*. 197–215. DOI: http://dx.doi.org/10.1007/978-3-642-28279-9_15

[10] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 163–172.

[11] Stefan Kögel, Raffaela Groner, and Matthias Tichy. 2016. Automatic Change Recommendation of Models and Meta Models Based on Change Histories. In *Proceedings of the 10th Workshop on Models and Evolution (ME2016), Saint-Malo, France, October 2, 2016*. 14–19. http://ceur-ws.org/Vol-1706/paper3.pdf

[12] Stefan Kögel and Matthias Tichy. 2017. Mining GIT Repositories for ECORE Models and their Version Histories. under submission.

[13] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. 2013. A posteriori operation detection in evolving software models. *Journal of Systems and Software* 86, 2 (2013), 551–566.

[14] Christoph Legat, Jens Folmer, and Birgit Vogel-Heuser. 2013. Evolution in Industrial Plant Automation: A Case Study. In *Annual Conference of the IEEE Industrial Electronics Society (IECON)*. Wien, ÃŰsterreich. ;

[15] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. *Introduction to recommender systems handbook*. Springer.

[16] Martin P Robillard, Walid Maalej, Robert J Walker, and Thomas Zimmermann. 2014. *Recommendation systems in software engineering*. Springer Science & Business.

[17] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons.

[18] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. 2010. Towards domain-specific model editors with automatic model completion. *Simulation* 86, 2 (2010), 109–126.

[19] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[20] Daniel Strüber. 2017. Generating Efficient Mutation Operators for Search-Based Model-Driven Engineering. In *International Conference on Theory and Practice of Model Transformations (ICMT)*. https://rgse.uni-koblenz.de/web/pages/research/papers/Str17.pdf

[21] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. 2017. Change-Preserving Model Repair. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 283–299.

[22] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[23] Andreas H Zins, Ulrike Bauernfeind, Fabio Del Missier, Adriano Venturini, and Hildegard Rumetshofer. 2004. *An experimental usability test for different destination recommender systems*. na.