Towards Performance Engineering of Model Transformation

Raffaela Groner, Matthias Tichy
Institute for Software Engineering and Programming Languages, Ulm University
Ulm, Germany
[raffaela.groner|matthias.tichy]@uni-ulm.de

Steffen Becker
Institute of Software Technology, University of Stuttgart
Stuttgart, Germany
steffen.becker@informatik.uni-stuttgart.de

ABSTRACT

Model transformations are an essential operation on models which is applied at design time and even at run time. For this, the performance of transformations is an important aspect, which needs to be considered. The current research takes only the improvement of transformation engines into account but there is no method or tool support to help engineers to identify performance bottlenecks in their transformation definition. In this paper we present our proposed approach to develop a method for performance engineering of model transformations. This method should support engineers to improve the performance of their defined transformations by providing visualizations of reasons for performance problems and offering possible refactorings for a transformation which can improve its performance.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software performance**;

KEYWORDS

Model Driven Software Engineering; Model Transformation; Performance Engineering; Henshin

ACM Reference Format:
Raffaela Groner, Matthias Tichy and Steffen Becker. 2018. Towards Performance Engineering of Model Transformation. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186305

1 INTRODUCTION

One trend to handle the continuous growth of the complexity of software systems is Model-Driven Engineering (MDE) which advocates using models as key artifacts. Model transformations are the second key artifact as they translate input models into output models. They are specified in transformation scripts executed by engines. As such, they enable the generation of new models, realizing changes on individual models, and the synchronization between models.

Models can become large, e.g., in the automotive domain and for performance predictions of information systems. For example, an AUTOSAR model of a large electronic control unit for a modern car has over 170.000 model elements. In such cases, the execution of badly performing model transformations can take up to hours. Hence, performance is an important quality of model transformations, e.g., throughput in terms of model elements or overall execution time.

Engineers today mostly address model transformation performance not at all, in an ad-hoc fashion, or after first problems arise in production. Various reasons for this exist: Current research addressing model transformation performance focuses on optimizing the engines internally with different heuristics (e.g., [6, 16]). However, this only optimizes each model transformation in a model transformation chain individually. Furthermore, transformation engineers are not supported in improving the transformation scripts themselves which in our experience leads to massive complementary performance gains. Unfortunately, this requires expert knowledge about how the model transformation engines interact with the transformation script to leverage these performance gains (cf. [12]).

Existing techniques addressing performance cannot be reused as-is and need to be adapted. Traditionally, profiling approaches enable inspecting systems during runtime to identify hot spots in which the program consumes resources excessively. Worst case execution time (WCET) analysis approaches compute worst case execution times for program parts. Both only work on a programming language level and, thus, are not reasonably applicable to model transformation performance. Software Performance Engineering (SPE) analyzes programs on a model level at design time to predict their performance and, thus, identify performance problems before they arise. However, existing approaches are not suitable for model transformations since they neither understand the heuristics of model transformation engines nor are able to analyze the performance impact of model transformation language features. Hence, the predictions can be completely wrong and, thus, useless. In summary, there exist no research activities to systematically and holistically enable the performance engineering of model transformations, i.e., to support the engineer in developing transformations that achieve the required performance right from the beginning.

Our goal is to provide a first-class environment for supporting software engineers in effectively improving the performance of model transformations. This evironment will enable software engineers to systematically *identify and visualize causes for performance issues* as well as *predict and improve the performance* of model transformation.

In the next section, we discuss in more detail the related work in performance and model transformations. Section 3 contains a presentation of our proposed approach which is illustrated using an example in Section 4. Thereafter, we conclude and give an outlook on our next steps.

2 RELATED WORK

While extensive state of the art exists in the general area of software and performance analysis, we focus in the following on the state of the art closely related to performance and model transformations.

Model transformations basically follow two different paradigms: operational and relational transformations (see [5] for a more detailed taxonomy). Operational model transformations follow the paradigm of imperative programming languages where the developer explicitly defines the different changes to the models and the control flow between them. Relational and graph-transformation based model transformations specify the changes in a declarative way and the transformation engines changes the model in such a way that the defined relations resp. graph patterns hold.

While operational model transformations are more similar to programming languages studied in performance engineering, they still differ in the way transformations are expressed and in the characteristics of models as inputs. Relational model transformations are quite different and the engine heuristics have a high impact on the resulting performance.

Research in performance optimization in model transformations primarily addresses transformation execution engines. As many different approaches exist, we only review representatives of the general classes.

A class of approaches, e.g. [21], statically analyze the model transformation and use heuristics for optimizing the average case execution time. Our own previous work [4, 11] does the same for the worst case execution time. Another line of research, e.g., [2, 6, 16] including the Henshin interpreter [12], additionally considers the model to execute the model transformation on and optimizes the search plan accordingly, e.g., using heuristics and cost models.

While the previous approaches address only the model transformation engine internally, some approaches exist to assess the performance of model transformations and improve it. A first area is the definition of metrics for model transformations [14, 17]. These papers' approaches mainly adapt metrics from programming languages to model transformations. While [14] is mainly concerned with maintainability metrics, only few of the metrics presented in [17] are related to performance.

Specifically for Henshin graph transformations, Taenzter et al. define bad smells and refactorings [10]. However, they do not address performance. Our own preliminary work [12], describes several performance related bad smells including detectors formalized as Henshin graph transformations. However, it is currently limited due to only statically analyzing the syntax and not integrating monitoring information from the execution.

Mészáros et al. present a concrete case [8] where they manually optimized a set of model transformations and achieved a 70% increase in performance. Similarly, Bruni and Lluch-Lafuente compare different ways to specify model transformations with respect to their performance in different benchmarks [3] and include some guidelines how to improve the performance by changing the model transformations. Wimmer et al. present a catalog of model transformation refactorings [18] which includes a few refactorings which improve and degrade the performance of the model transformations in their experimental evaluation.

While all these approaches show that significant performance gain can be achieved by refactoring the model transformations done by expert engineers (with extensive knowledge of the model transformation engine), a systematic process to support the normal engineer is currently lacking.

3 PROPOSED APPROACH

Our goal is to develop an approach which helps engineers to improve the performance of their implemented transformations by predicting the performance of model transformation executions, visualizing causes for performance bottlenecks and offering refactorings that influence the performance of a transformation positively.

We plan an iterative approach, which consists of five work packages: **Monitoring**, **Predict**, **Analyze**, **Visualize**, and **Improve**. This approach is similar to the steps presented in [7] for a performance evaluation study and also contains software performance engineering activities mentioned in [20]. After those five work packages we plan to use different evaluation techniques like proof-of concepts or empirical evaluation to guarantee that the interplay of the single results realize our overall goal. Therefore, we plan to develop a benchmark suite and we will evaluate the complete approach with users.

Monitoring: To analyze the performance of model transformations it is necessary to identify and understand the properties that affect the performance of a transformation. Those properties can be e.g. the input model complexity, like the number of objects and the number of different types of objects a model contains, their in- and out-degree, the resource demands of the execution of a transformation or parameters of the transformation engine. With the help of the framework Kieker ([15]) and our own monitoring infrastructure we collect raw data, that we need for the further proceeding.

Predict: Based on the raw data from **Monitoring** we want to develop a prediction framework. This framework should support an engineer to predict the performance change of a model transformation execution based on changes in the input model, in the model transformation, in the engine's infrastructure or by exchanging the engine heuristic strategy. With the help of this framework an engineer should be able to predict e.g the performance changes of a transformation execution, if the number of objects in the input model is changed.

Analyze: The obtained raw data from work package **Monitoring** will be too detailed to help a transformation engineer to identify the causes for performance problems. For this, we want to develop in this work package an analysis approach which identifies and ranks possible causes for performance problems. Corresponding to code bad smells model transformations can also contain bad smells, which influence the performance negatively [13].

Visualize: To improve the performance of a transformation it is important to understand in which way the transformation is executed, why is it executed in that way and what are the factors influencing the performance of this execution. To help a transformation engineer to understand those aspects we plan to develop visualizations for causes like properties of the input model, engine decisions and performance results. Such visualizations could be a blame graph, which shows the possible causes for performance problems or a scatter plot, which relates the execution time of a transformation on a model to the number of its objects.

Improve: In this package, we plan to develop refactorings, which improve the performance of model transformations. Those suggestions for improvement should help to fix the causes of performance problems from work package **Analyze**, as well as refine and extend the performance improving refactorings from literature like [19] and [9] for previous identified performance problems. A suggestion for improvement for a transformation, which contains a bad smell, could be a refactoring. The performance gain of a suggestion for improvement depends also on the context of the transformation application. For this, we plan to investigate those dependencies and develop guidelines, when a suggestion for improvement can be applied as well as how and why it improves the performance.

4 ILLUSTRATIVE EXAMPLE

We illustrate our proposed approach with the help of an example consisting of a model transformation in Henshin. Henshin is a model transformation language, which is based on graph transformation concepts. A transformation is defined by a rule which consists of two attributed, typed graphs, called left hand side (LHS) and right hand side (RHS). The LHS defines the precondition and describes how a part of the model has to be defined to apply the transformation. The RHS defines the postcondition and describes how the model is changed when the rule is executed [1]. Our illustrative transformation rule is called *transferMoney* and is shown on the top of Figure 1. This representation of the rule combines the LHS and RHS and annotates the pre- and postcondition. This transformation rule transfers an amount of money, that is defined by *amount*, from one account with *credit x*, which is referred by an map entry (*EInt2AccountMap*) with *key fromId*, to another account with *credit y*, which is referred by an map entry with the the value *toId* for its *key*. *amount*, *fromId* and *toId* are parameters which are set by the user before the transformation is executed. x and y are variables which serve as place holders for the real values assigned to *credit*.

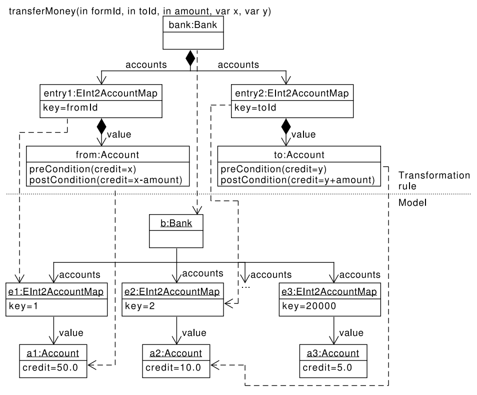

Figure 1: Matching of the transformation rule *transferMoney*

In our example we applied the transformation *transferMoney* with *amount*=1, *fromId*=1 and *toId*=2 on the model in the bottom of Figure 1. This model consists of one *Bank* instance and 20000 *Account* instances, which are connected by corresponding *EInt2AccountMap* instances. First, a search plan is created, that determines the order in which for each node of the LHS a strategy is derived how to find a model element with the same type, the same references and the same attribute values. If such a model element is found its called a match. The search plan for this example is $\{entry1, entry2, bank, from, to\}$. To find a match for *entry1*, all model elements with the same type, here $\{e1, e2, ..., e20000\}$, are candidates. The Henshin engine iterates this list in reverse order and investigates each element if its attribute *key* has the value *fromId*. In this example the engine investigates *e20000* to *e1* and determines *e1* as match for the node *entry1*. Afterwards the engine tries to find in the same way a match for *entry2* and determines *e2*. Then a match for *bank* is searched in the same way. Because the matched model element for *bank* needs to refer the matched model elements for *entry1* and *entry2*, the engine does not only consider the type and the attribute values but also the existence of those references. Our example model only has one instance of type *Bank* and this refers all *EInt2AccountMap* instances, so this *b* is the match for *bank*. Next the engine searches a match for *from*. The LHS defines that the match for *from* needs to be referred by the match for *entry1*. Thus the list of candidates for *from* is reduced to all instances of *Account*, which are referred by the match for *entry1* . Here, this means only *a1* has to be investigated. This model element is a match for *from* because x is only a place holder that doesn't define a constraint. At last a match for *to* is searched in the same way. The found matching is illustrated in Figure 1 as dashed arrows.

We repeated the application of the rule *transferMoney* 100 times on the same input model, with *amount*=1 and randomized for each application the values of *fromId* and *toId*. In the package **Monitoring**, we monitor the matching order of the nodes from the LHS, which is defined by the search plan, and the duration until a match for each node was found. We also monitor for each node of the LHS the number of model elements, which were investigated until one was found that matches this node. In our first simple **Analyze** phase, we use our knowledge about how the order in the search plan can influence the number of model elements which need to be investigated. E.g. we described above that the candidates for *from* are reduced because they need a reference from *entry1*. If *entry1* wasn't matched at this time all instances of *Account* have to be investigated. For this, we developed in the **Visualize** package the visualizations for our example as shown in Figure 2. The x-axis of this box plot shows the order of the nodes in the search plan, here $\{entry1, entry2, bank, from, to\}$. The y-axis shows the number of model elements, which are investigated until a match was found. With the help of this visualization, a transformation engineer gains the insight that during the matching of the map entries *entry1* and *entry2* a lot of model elements are investigated to find a match for them. This leads to a duration of on average 10,8ms until a match for the whole LHS was found.

Based on our expert knowledge of the Henshin engine, we knew that the automatically derived search plan is bad, since the entry nodes *entry1* and *entry2* are matched first. An improvement

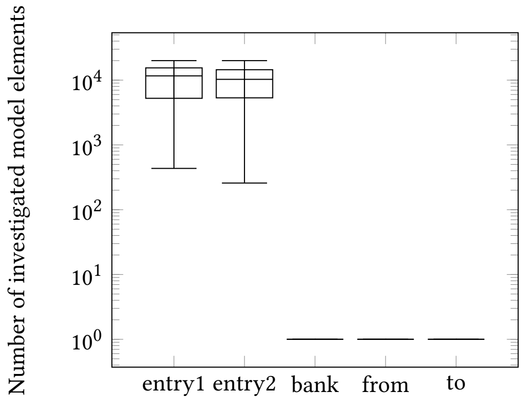

Figure 2: Application of *transferMoney*

(package **Improve**) is to change the order in the search plan to $\{bank, entry1, entry2, from, to\}$. Because if the node *bank* is matched first the Henshin matching engine identifies that the nodes *entry1* and *entry2* are map entries and uses a map to lookup the accounts *from* and *to*. The guideline for this improvement is that nodes like *entry1* and *entry2*, which are map entries should be matched after the node that refers them. Figure 3 shows the performance measurements of the improved application of the rule *transferMoney*. It clearly shows that the change of the order reduces the number of investigated model elements. Therefore, the duration of finding a match for *transferMoney* drops to an average of 2,56ms.

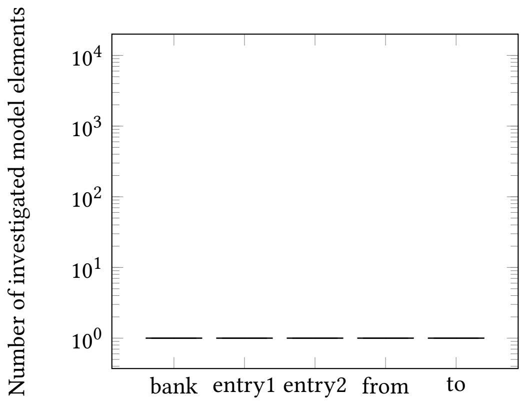

Figure 3: Optimized application of *transferMoney*

5 CONCLUSION AND FUTURE WORK

The performance of model transformations is an important factor, but at the moment there is no actual support for engineers to improve their transformations. We propose an approach to solve this problem consisting of a combination of the packages **Monitoring**, **Predict**, **Analyze**, **Visualize** and **Improve** and illustrate these, excluding **Predict**, with an example. Currently, we are working on the realization of our work packages. In order to evaluate the usability and the effectiveness of our approach we will perform user studies with transformation engineers.

6 ACKNOWLEDGEMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 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. *Model Driven Engineering Languages and Systems* (2010), 121–135.
[2] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. 2008. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) (LNCS)*, Vol. 5088. Springer. http://www.info.uni-karlsruhe.de/papers/agtive_2007_search_plan.pdf
[3] Roberto Bruni and Alberto Lluch-Lafuente. 2011. Evaluating the Performance of Model Transformation Styles in Maude. In *FACS (Lecture Notes in Computer Science)*, Vol. 7253. Springer, 79–96.
[4] Sven Burmester, Holger Giese, Andreas Seibel, and Matthias Tichy. 2005. Worst-Case Execution Time Optimization of Story Patterns for Hard Real-Time Systems. In *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany.* 71–78.
[5] Krzysztof Czarnecki and Simon Helsen. 2006. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45, 3 (2006), 621–645.
[6] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. 2009. Improved Flexibility and Scalability by Interpreting Story Diagrams. *ECEASST* 18 (2009). http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/268
[7] Raj K Jain. 1991. The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling. (1991).
[8] Tamás Mészáros, Gergely Mezei, Tihamer Levendovszky, and Márk Asztalos. 2010. Manual and automated performance optimization of model transformation systems. *STTT* 12, 3-4 (2010), 231–243.
[9] Tamás Mészáros, Gergely Mezei, Tihamér Levendovszky, and Márk Asztalos. 2010. Manual and automated performance optimization of model transformation systems. *International Journal on Software Tools for Technology Transfer (STTT)* 12, 3 (2010), 231–243.
[10] Gabriele Taentzer, Thorsten Arendt, Claudia Ermel, and Reiko Heckel. 2012. Towards refactoring of rule-based, in-place model transformation systems. In *Proceedings of the First Workshop on the Analysis of Model Transformations (AMT '12).* ACM, New York, NY, USA, 41–46. https://doi.org/10.1145/2432497.2432506
[11] Matthias Tichy, Holger Giese, and Andreas Seibel. 2006. Story Diagrams in Real-Time Software. In *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany (Technical Report)*, Holger Giese and Bernhard Westfechtel (Eds.), Vol. tr-ri-06-275. University of Paderborn, 15–22.
[12] Matthias Tichy, Christian Krause, and Grischa Liebel. 2013. Detecting performance bad smells for Henshin model transformations. In *Proc. of the 2nd Workshop on the Analysis of Model Transformations (AMT), September 29, Miami, USA*, Benoit Baudry, Jürgen Dingel, Levi Lécio, and Hans Vangheluwe (Eds.).
[13] Matthias Tichy, Christian Krause, and Grischa Liebel. 2013. Detecting Performance Bad Smells for Henshin Model Transformations. *AMT@ MoDELS* 1077 (2013).
[14] Marcel van Amstel, Mark van den Brand, and Phu H. Nguyen. 2010. Metrics for Model Transformations. In *BENEVOL 2010 (9th Belgian-Netherlands Software Evolution Seminar, Lille, France, December 16, 2010. Proceedings of Short Papers).* Université Lille 1, 1–5.
[15] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012).* ACM, 247–248.
[16] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. 2015. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Software and System Modeling* 14, 2 (2015), 597–621. https://doi.org/10.1007/s10270-013-0372-2
[17] Andrés Vignaga. 2009. *Metrics for Measuring ATL Model Transformations.* Technical Report TR/DCC-2009-6.
[18] Manuel Wimmer, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. 2012. A Catalogue of Refactorings for Model-to-Model Transformations. *Journal of Object Technology* 11, 2 (Aug. 2012), 2:1–40. https://doi.org/10.5381/jot.2012.11.2.a2
[19] Manuel Wimmer, Salvador Martínez Perez, Frédéric Jouault, and Jordi Cabot. 2012. A Catalogue of Refactorings for Model-to-Model Transformations. *Journal of Object Technology* 11, 2 (2012), 2–1.
[20] Murray Woodside, Greg Franks, and Dorina C Petriu. 2007. The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07.* IEEE, 171–187.
[21] Albert Zündorf. 2002. *Rigorous Object Oriented Software Development.* University of Paderborn.