

Comparing Reuse Mechanisms for Model Transformation Languages: Design for an Empirical Study

Daniel Strüber
Universität Koblenz-Landau, Germany* and
Philipps-Universität Marburg, Germany
strueber@uni-koblenz.de

Anthony Anjorin
University of Paderborn, Germany
anthony.anjorin@upb.de

ABSTRACT

Reuse mechanisms for model transformation languages help avoid duplication, thereby increasing maintainability and enabling a more systematic overall development process. As the introduction of such reuse mechanisms to model transformation languages is still in its preliminary stages, however, language designers are currently faced with the challenge of choosing from amongst diverse proposed approaches. Although there are a few surveys comparing syntactic and semantic differences, there is still a need for empirical studies on the actual effectiveness of different reuse mechanisms for transformation developers. In this paper, therefore, we present a study design for a controlled experiment to investigate the benefits and drawbacks of two specific reuse mechanisms for model transformation languages: rule refinement and variability-based rules. Both mechanisms are tailored to graph-based model transformation languages, yet represent two contrasting reuse paradigms: *modularizing* rules by composing them from smaller, shared fragments, versus maintaining a single, integrated representation via *variability annotations*. We propose to compare these two approaches using comprehension and bug-finding tasks to investigate understandability, and bug-fixing and modification tasks to study changeability.

1. INTRODUCTION

Model transformations play a fundamental role in Model-Driven Engineering (MDE), where software systems are produced by the continuous refinement, translation, and synchronization of models [20].

As the field matures and MDE approaches are faced with increasingly complex development tasks, the size and complexity of required model transformations also increases. Developing and especially maintaining such large transforma-

*This research was partially supported by the research project *Visual Privacy Management in User Centric Open Environments* (supported by the EU's *Horizon 2020* programme, Proposal number: 653642).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HuFaMo '16 October 4, 2016, Saint Malo, France

© 2016 ACM. ISBN 123-4567-24-567/08/06...\$15.00

DOI: 10.475/123_4

tions is challenging without dedicated support for reusing parts of existing transformations [23]. As such reuse mechanisms for model transformation languages are just starting to emerge [15], however, transformation developers are often forced to resort to code or pattern duplication, well-known for its manifold maintainability-related drawbacks.

But how can language designers decide which reuse mechanisms to integrate in their model transformation languages? While some surveys exist comparing syntactic and semantic differences of various proposed reuse mechanisms (cf., e.g., [15]), there exist no empirical studies of various transformation reuse mechanisms and their actual usefulness for transformation developers.

In this work, therefore, we present our design for a controlled experiment that aims to explore the benefits and drawbacks of the proposed reuse mechanisms for model transformation languages. Specifically, we intend to compare two recently emerging reuse mechanisms: rule refinement [4] and variability-based rules [25]. This choice is motivated by the fact that both mechanisms address the same technical scope - the reuse of graph-based model transformations [7] - while contrasting in the employed reuse *paradigm*: rule refinement reuses transformation rules by breaking them into smaller fragments that can be assembled using a composition mechanism, while a variability-based rule represents a family of multiple, similar rules in an integrated form, using variability annotations to denote individual parts. This distinction between composition- and annotation-based approaches is well-known from the domain of software product lines [12].

We are interested in the influence of these reuse mechanisms on *maintainability*, a main quality aspect of software artifacts [1] that is particularly important in the face of continuous development and evolution. More specifically, our study targets the maintainability aspects of *understandability* and *changeability* of model transformations [19, 27]. Given a group of participants, understandability can be quantified in terms of correctness, completion time, and perceived effort during bug-finding and multiple-choice style comprehension tasks. To measure changeability, we plan to use bug-fixing and modification tasks, again using correctness, completion time, and perceived effort for quantification.

The rest of the paper is structured as follows: Section 2 discusses related work. Section 3 introduces a motivating running example that is used to explain the two reuse mechanisms for the comparison. Section 4 presents our study design with an outline of the experimental variables of the set-up, while Section 5 concludes.

2. RELATED WORK

The planned study is the first empirical investigation of transformation reuse mechanisms. Empirical studies in the area of model transformation have so far been confined to the evaluation of transformation languages and approaches. Acretoia et. al [2] have compared their usability-oriented Visual Model Transformation Language to traditional visual and textual languages. They found that their approach establishes a compromise between effort and correctness. In an evaluation of their transformation-by-example approach, Avazpour et al. [6] found that developers perceive the provided interaction-based tool support as useful during the development of transformations. Guana and Stroulia [10] propose a study design to understand developer performance during the development of model-to-code transformations and measure the effect of traceability visualizations. Conversely, only reuse mechanisms for code have been investigated so far, mostly from a product line perspective [9, 16].

3. MOTIVATING SCENARIO

Our motivating scenario (adapted from [24]) is a simplified *move method* refactoring, implemented via three transformation rules depicted in Fig. 1: The first rule A transfers a method *m* from its current containing class *s* to another class *t*. This rule is a *graph transformation rule* that preserves all elements marked with **preserve**, deletes all elements marked with **delete**, and creates all elements marked with **create**. Such rules are only applicable if all preserved and deleted elements can be matched in a host model.

Rules B and C represent variants of A: B creates a wrapper method *w* for simple delegation to the method *m* that is to be moved. This can be used to limit the effects of the refactoring and avoid having to change all existing clients of the class *s*. Rule C additionally creates an annotation that marks *w* as deprecated to discourage future usage of the wrapper method, which can be eventually deleted.

This simple example illustrates how multiple transformation rules can “share” a common part that must be repeated in each variant if no reuse mechanism is provided. In realistic applications where possibly hundreds of rules are required, such pattern duplication can quickly lead to a maintenance nightmare as changes, e.g., bug fixes, have to be consistently applied to multiple, independent rules without tool support.

3.1 Rule refinement

The basic idea of rule refinement [4] is to enable a flexible sharing and composition of common rule fragments. Pattern duplication can be avoided in this way by including the same fragment as a part of multiple rules. Rule refinement can be applied to our running example to result in a network of rule fragments as depicted in Fig. 2. Rule A remains unchanged and is regarded as “resolved” as it does not refine any other rule fragment (it has no outgoing “inheritance” arrows). Rule B is reduced to a refinement of A, i.e., it is produced by merging rule A with the specified refinement. This merge operator is relatively straightforward: The disjoint union of both patterns (in general of multiple patterns) is formed, and then all elements with the same identifier are glued together. Similarly, rule C is specified as a refinement of B, resulting in the same pattern as in Fig 1 by first resolving rule B and then repeating the same merging process with the specified refinement for rule C. Note that the merge

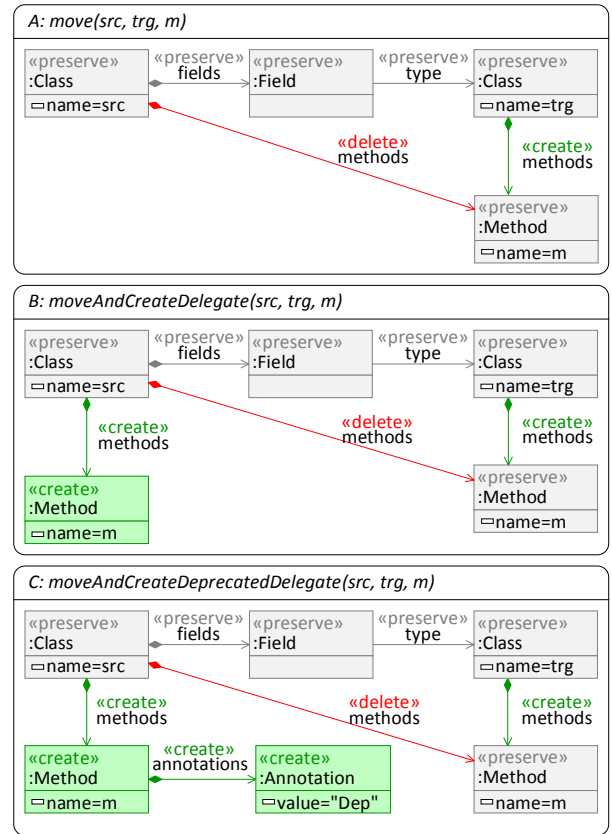


Figure 1: Three model transformation rules.

operator requires the manual specification of identical elements, which is done by assigning *names* (e.g., *s*, *me*) to pattern variables, a concept that is not to be confused with *parameters* (e.g., *src*, *trg*, *m*), which are used to propagate values across name attributes in this example.

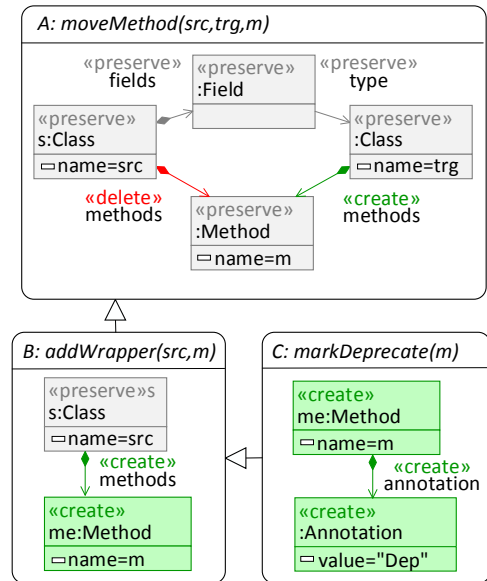


Figure 2: Refinement rules.

3.2 Variability-based rules

In stark contrast to rule refinement, *variability-based* rules [22, 24, 25] enable an integrated representation of multiple rules in a single diagram. The variability-based rule for our running example is depicted in Fig. 3. The rule comprises two configuration options that are denoted via variability annotations, e.g., the annotation `[wrapper]` is used to indicate which edges and nodes in the diagram are only present when configuration option `wrapper` is set to true. Note that dependencies between configuration options can be specified, e.g., the dependency `deprecate -> wrapper` (upper-left corner) is used to indicate that the configuration option `deprecate` requires `wrapper`, i.e., all elements annotated with `wrapper` are also present in the `deprecate` variant of the rule.

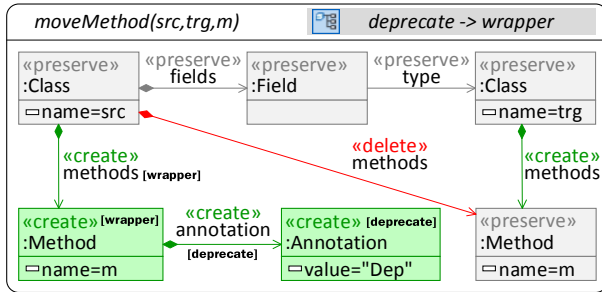


Figure 3: Variability-based rule.

This scenario illustrates a set of maintainability trade-offs that we aim to explore in our study. Both considered mechanisms help avoid pattern duplication, which may simplify the consistent editing of multiple variants at once. In addition, if applied skillfully, rule refinement can lead to comprehensible and maintainable modular rule fragments, reflecting an independent handling and separation of concerns in the transformation. Variability-based rules improve notational compactness drastically, thereby potentially reducing reading effort: it is well-known that humans struggle with processing more than 7 ± 2 units of information at once [18]. On the downside, as they require developers to derive the included rule variants in their minds, both approaches may increase cognitive overhead. In particular, variability-based rules may suffer from their increased amount of information in each individual diagram, comparable to the case of variability-annotated code, where a decrease in developer performance was found [16].

4. STUDY DESIGN

To investigate the identified trade-offs further, we propose the following research questions:

- **RQ1:** Which reuse mechanism is most beneficial for developers in terms of understandability and changeability?
- **RQ2:** How do the studied reuse mechanisms compare to offering no reuse at all?
- **RQ3:** Can the understandability and changeability of the selected reuse mechanisms be further improved via the available tool support?

To address these research questions, we plan to administer a set of comprehension and implementation tasks to participants who are familiar with graph-based model transformations. Such participants will be recruited from research groups, as well as master-level model-driven engineering courses.

We plan to apply a *crossover* trial, a variant of the *within-subject* design [11]: in this design, all subjects are sequentially exposed to each treatment. This choice is motivated by the statistical efficiency of the design: it allows the number of required participants to be minimized. Moreover, the influence of confounding factors (e.g., participant expertise) is reduced as each participant serves as their own control. A main threat to this kind of study design are learning effects: over the course of the experiments, participants become more experienced and might carry over gained experience among tasks. A viable mitigation to this threat is to vary the order of treatments among participants. We discuss additional threats and mitigation strategies in Sec. 4.3.

To discuss the different dimensions that influence our experiment, we first outline its variables in the following. We then specify our research plan as one possible path through the design space spanned by these variables. Finally, we describe potential threats to validity in our set-up, and our means of mitigating these threats.

4.1 Experimental variables

4.1.1 Independent variables

As the single independent variable in the planned experiment, we measure the effects of the employed reuse mechanism. For RQ1 and RQ3, only the data points of the two proposed reuse mechanisms are considered, whereas RQ2 incorporates the data points for the no-reuse case as well.

4.1.2 Dependent variables

The variables we want to observe as a result of the experiment are the following:

Correctness score. We plan to measure correctness based on how well participants are able to solve certain tasks. As these tasks might not be directly comparable, counting the number of correctly solved tasks only provides ordinal values.

Perceived effort. Participants will be asked to rate the perceived effort it took to solve all tasks using a suitable ordinal scale ranging from, e.g., 0 to 10 to quantify effort.

Completion time. For all tasks, completion times will be measured for all participants. This qualitative value can be contrasted with correctness scores and perceived effort to yield further insights.

User preference. Finally, regardless of correctness, perceived effort, and completion time, we shall ask users which reuse mechanism they prefer and enjoyed using the most.

For statistical analysis, we intend to apply the Wilcoxon signed-rank test [29], which is a robust non-parametric test to identify significant differences between two treatments. It assumes data measured at least on an ordinal scale, which applies to all of our dependent variables.

A remaining issue concerns the fact that we are interested in two different quality goals that should ideally be studied as separate entities. Yet, as changeability clearly depends on understandability, it is not obvious how these two different goals can be studied in isolation. To this end, a viable

	Name	Possible values
Independent variable	Reuse mechanism	{ Rule refinement, variability-based rules, no reuse }
Controlled variables	Transformation language	{ Henshin, ATL, EOL, ... }
	Example type	{ queries, endogenous, exogenous }
	Task type	{ bug-finding, quiz, bug-fixing, modification task }
	Environment	{ pen-and-paper, tool }
Dependent variables	Correctness score	Ordinal
	Perceived effort	Ordinal
	Completion time	Interval
	User preference	Ordinal

Table 1: Experimental variables

method is to collect performance data for both goals, and to subtract the understandability from the changeability results. For instance, the time required to actually change the affected rules can be obtained from subtracting the time required to understand the rules from the full completion time. A prerequisite for this method is that subtraction is supported by the underlying scale, which is the case for completion times.

4.1.3 Controlled variables

The design space of our controlled variables comprises the following dimensions:

Transformation language. The variety of available model transformation languages is usually classified along a number of main paradigms, most prominently imperative, hybrid, and graph-based model transformations [?, 17]. In the planned study, we aim to investigate reuse in the context of Henshin [5], a graph-based language. The reason for this choice is twofold: first, examples and tool support for one of the supported reused mechanisms are readily available in Henshin (convenience sampling). Second, the graphical syntax of Henshin closely resembles that of a group of languages including eMoflon [3], EMorF [13], and QVTs [30], rendering it a promising choice to obtain results which might also generalize to these languages. In addition, Henshin is under active development and in widespread use.¹ Our study will be based on bug-finding and bug-fixing tasks, since these tasks can avoid lengthy textual task descriptions that might be hard to match to the presented visual representations [28]. Our example rules will mostly be endogenous, such as those shown in the running example, allowing us to establish a compromise between external and internal validity.

Example types. In the choice of example transformations, we need to account for the inherent trade-off between external and internal validity [21]: by using simple and easy-to-understand examples, we can mitigate the effect of participant heterogeneity, but risk ending up with an experiment that is not realistic enough. The simplest transformation that can be expressed using rules is a *query* – a rule performing no modifications of the input model. Queries are generally easy to understand; yet in realistic settings they are likely to be expressed using dedicated query languages. Endogenous transformations – transformations that update the input model or translate it to a model of the same language – are a typical use-case for graph-based model transforma-

tion, but require an understanding of its underlying replacement mechanism. *Exogenous transformations*, creating an output model of a different language, are most complicated as they require the understanding of two meta-models and their relations; yet they are particularly important for the typical forward-engineering process in model-driven engineering. Two other sub-dimensions of this dimension are the number of examples and the size of the individual examples.

We intend to conduct a pilot study to find a suitable collection of examples that avoids participant fatigue, using a different set of participants than in the actual study to avoid history effects. Other goals of the pilot study are to determine if we can extract any meaningful results (e.g. maybe all task are too difficult and we do not gather sufficient useful data), and to see if our assumptions about participant’s abilities hold true.

Task types. To study understandability and changeability, we plan to administer tasks allowing us to quantify these properties. The understandability of a software artifact can be investigated using bug-finding tasks or quiz-style comprehension questions. In the example, a bug-finding task would be to swap the *delete* and *create* edges and have the user find and mark this error on a printout. A comprehension question could be “*How many individual rules can be derived from this representation?*”. Changeability refers to the ease of modifying or extending the functionality of a system. Two possible kinds of tasks to study changeability are the fixing of a preconceived bug, such as the one described above, and the implementation of a feature request such as “*I need a set of transformation rules with the same behavior as the given ones, but with one modification: the target class must have a field whose type is the source class*”. A challenge for the study design lies in choosing a feasible subset of all tasks so as to avoid exhausting the participants. To this end, we plan to use bug-finding and bug-fixing tasks, since these tasks can avoid lengthy textual task descriptions that might be hard to match against the presented visual representations [28].

Environment. Another balancing act between control and remaining adequately realistic concerns the experimental environment. By setting up the participants in a controlled pencil-and-paper environment, we can eliminate confounding factors related to the tool implementations, for instance, differences in visual appearance. Such a controlled environment is, however, not really comparable to the real-world environment that actual users of the reuse mechanisms find themselves in, especially in the case of changeability tasks. Moreover, RQ3 can only be studied by comparing the

¹For a selection of publications describing Henshin’s use in numerous research projects, see <https://www.eclipse.org/henshin/publications.php>

reuse mechanisms with and without the actual tool support provided. As of now, tool support for rule refinement [14] and VB rules [26] exists, but is implemented within different model transformation languages and corresponding tool environments – eMoflon and Henshin. To ensure optimal control during the experimental comparison, both mechanisms should be implemented for the same language and tool environment.

4.2 Quantitative data

To gain additional insight about the perceived benefits and pitfalls of both reuse approaches, we also plan to collect quantitative data, using text fields in the task sheet to collect participants' opinions after their completion of the experiment.

4.3 Potential threats to validity

In addition to the threats already described in the previous discussion, we plan to mitigate *researcher bias* by striving to ensure that participants are unable to directly map reuse mechanisms to researchers, e.g., by anonymising tool names.

To avoid selection bias in favor of volunteers, whose level of motivation does not reflect that of the whole population, we will offer a small prize among participants.

Finally, the participants' performance can be confounded by relevant prior knowledge, e.g., from exercise in the area of software product lines and other annotation-based model families. To study the effect of prior knowledge, we plan to ask participants to rate their own prior knowledge of relevant topics.

5. CONCLUSION

In this paper, we present our study design for an empirical comparison of two reuse mechanisms for graph-based model transformation languages. The selected reuse mechanisms exemplify the contrasting reuse paradigms of *modularizing* a set of rules and merging similar rules in a *variability-annotated* representation.

Our prior expectations are as follows: we expect refinements to be (perhaps only slightly) more understandable but less changeable than variability rules (RQ 1), both reuse mechanisms to substantially improve changeability but (at least without substantial training) reduce understandability (RQ 2), and that appropriate tool support plays a crucial role in both cases (RQ 3). It remains to be seen if our experiment results will support or refute these initial hypotheses. Addressing the trade-off between different forms of experimental validity [21], our set-up aims to establish a compromise between internal and external validity. A series of follow-up studies moving the spotlight in either direction can lead to a more comprehensive picture.

The results of our study will provide evidence to designers of model transformation languages who are planning to include a reuse mechanism within their language. In a broader context, we hope to shed a new light on the discussion of when and how software reuse can be established in a manner that is most beneficial to all involved stakeholders.

Acknowledgements: The authors wish to thank Vlad Acrețoiaie and the anonymous reviewers for their careful reading of this paper and their detailed comments and suggestions.

6. REFERENCES

- [1] ISO/IEC 25010:2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.
- [2] V. Acrețoiaie, H. Störrle, and D. Strüber. VMTL: a language for end-user model transformation. *Software & Systems Modeling*, pages 1–29, 2016.
- [3] A. Anjorin, M. Lauder, S. Patzina, and A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. *Informatik*, 192:281, 2011.
- [4] A. Anjorin, K. Saller, M. Lochau, and A. Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In *International Conference on Fundamental Approaches to Software Engineering*, pages 340–354. Springer, 2014.
- [5] 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.
- [6] I. Avazpour, J. Grundy, and L. Grunske. Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations. *Journal of Visual Languages and Computing*, 28:195 – 211, 2015.
- [7] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
- [8] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachsel, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, 18(4):699–745, 2013.
- [9] V. Guana and E. Stroulia. How Do Developers Solve Software-engineering Tasks on Model-based Code Generators? An Empirical Study Design. In *First International Workshop on Human Factors in Modeling*, pages 33–38. CEUR-WS.org, 2015.
- [10] B. Jones and M. G. Kenward. *Design and analysis of cross-over trials*. CRC Press, 2014.
- [11] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *International Conference on Software Engineering*, pages 311–320. ACM, 2008.
- [12] L. Klassen and R. Wagner. EMorF-A tool for model transformations. *Electronic Communications of the EASST*, 54, 2012.
- [13] G. Kulcsár, E. Leblebici, and A. Anjorin. A Solution to the FIXML Case Study Using Triple Graph Grammars and eMoflon. In *Transformation Tool Contest*, pages 71–75. CEUR-WS.org, 2014.
- [14] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *Software & Systems Modeling*, 14(2):537–572, 2015.
- [15] J. Melo, C. Brabrand, and A. Wařowski. How Does the Degree of Variability Affect Bug Finding? In *International Conference on Software Engineering, ICSE '16*, pages 679–690, New York, NY, USA, 2016. ACM.

- [16] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
- [17] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.
- [18] S. Nalchigar, R. Salay, and M. Chechik. Towards a catalog of non-functional requirements in model transformation languages. In *Workshop on the Analysis of Model Transformations*. CEUR-WS.org, 2013.
- [19] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.
- [20] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *International Conference on Software Engineering*, volume 1, pages 9–19. IEEE, 2015.
- [21] D. Strüber. *Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems*. PhD thesis, University of Marburg, 2016.
- [22] D. Strüber, T. Kehrer, T. Arendt, C. Pietsch, and D. Reuling. Scalability of Model Transformations: Position Paper and Benchmark Set. In *Workshop on Scalable Model Driven Engineering*, pages 21–30. CEUR-WS.org, 2016.
- [23] D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger. RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules. In *International Conference on Fundamental Approaches to Software Engineering*, pages 122–140. Springer, 2016.
- [24] D. Strüber, J. Rubin, M. Chechik, and G. Taentzer. A Variability-Based Approach to Reusable and Efficient Model Transformations. In *International Conference on Fundamental Approaches to Software Engineering*, pages 283–298. Springer, 2015.
- [25] D. Strüber and S. Schulz. A Tool Environment for Managing Families of Model Transformation Rules. In *International Conference on Graph Transformation*, pages 89–101. Springer, 2016.
- [26] E. Syriani and J. Gray. Challenges for addressing quality factors in model transformation. In *International Conference on Software Testing, Verification and Validation*, pages 929–937, 2012.
- [27] I. Vessey and D. Galletta. Cognitive fit: An empirical study of information acquisition. *Information systems research*, 2(1):63–84, 1991.
- [28] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, Dec. 1945.
- [29] E. D. Willink. Optimized Declarative Transformation: First Eclipse QVTc Results. In *Workshop on Scalable Model Driven Engineering*, pages 47–56. CEUR-WS.org, 2016.