Detecting Performance Bad Smells for Henshin Model Transformations

Matthias Tichy[1], Christian Krause[2], and Grischa Liebel[1]

[1] Software Engineering Division, Chalmers | University of Gothenburg, Sweden
matthias.tichy@cse.gu.se, grischa@chalmers.se
[2] SAP Innovation Center Potsdam, Germany
me@ckrause.org

Abstract. In model-driven software engineering, model transformations are used for the specification of model changes. Similar to programs also model transformations can exhibit bad smells which indicate possible weaknesses. In this paper, we address bad smells which can negatively affect the performance of the application of model transformations, particularly, model transformations defined in Henshin. Based on a description of the Henshin interpreter and its performance enhancing strategies, we describe a set of bad smells and corresponding detectors. We evaluate the detectors by applying them to the example rule set of Henshin.

Keywords: model transformation, Henshin, bad smell, performance, subgraph matching

1 Introduction

Model transformations are a key part of model-driven software engineering approaches as they define the changes on individual models or between models. There exist many different languages for the specification of model transformations, e.g., PMT [12], ATL [7], JTL [3], QVT, and the graph transformation based languages PROGRES [19], AGG [10], Fujaba [4], VIATRA2 [16] and Henshin [1].

Graph transformation based approaches, like the ones above, essentially define model transformations using rules consisting of a pre-condition graph, called the left-hand side (LHS), and a post-condition graph, called the right-hand side (RHS) of the rule. Informally, the execution of a model transformation requires that a matching of objects in the model (host graph) to the nodes and edges in the LHS is found and these matched objects are changed in such a way that the nodes and edges of the RHS match these objects.

The performance of graph transformation based model transformations are mainly determined by the efficiency of the match finding of the LHS. Consequently, model transformation languages offer different options to add constraints to the LHS of model transformations to improve the performance of the matching. To be efficient, graph transformation tools usually employ heuristics such as search plans to provide good performance (e.g. [14]). Henshin is a graph transformation based model transformation tool, which is tightly integrated into the Eclipse Modeling

Framework (EMF). Henshin does not enforce restrictions for the specification of the LHS in order to be as widely applicable as possible. Thus, software developers might not constrain their rules enough which might negatively affect the rule's execution performance for larger models.

The contribution of this paper is an informal description of bad smells in Henshin transformation rules which might negatively affect the execution performance. These bad smells are indicators for potential performance issues to be expected at runtime. Detectors of these bad smells are then formalized as Henshin rules (and in some cases utility code) on the Henshin meta model itself. Finally, we report on the amount of bad smells found in the examples provided with Henshin itself. We focus in this paper on static analysis on the transformation rules.

We introduce Henshin and its matching approach in the next section. Section 3 contains the description and formalization of the bad smells. The results of the execution of the bad smells detectors on the Henshin examples are shown in Section 4. After a short discussion of related work in Section 5, we conclude and present an outlook on future work in dynamic performance analysis of Henshin transformations.

2 Henshin

Henshin is a high-level graph rewriting and model transformation language and tool targeting models defined in the Eclipse Modeling Framework (EMF) [9]. Having its roots in academia, Henshin is also used in productive industrial settings. For instance, the tool is used by the internationally operating satellite provider SES to translate control program code for satellites (see [6] for details). The graphs in this application consist of hundreds of thousands of nodes. Therefore, a high performance of the graph pattern matching and transformation engine is crucial for practical use.

2.1 Graph transformation rules in Henshin

The Henshin transformation language is defined by means of a meta model. Figure 1 shows a part of this meta model which is relevant for this paper. Graph transformation rules are modeled using instances of the Rule class. The Henshin meta model is closely aligned to the underlying formal model of double pushout (DPO) graph transformations. Thus, rules consist of a left-hand side and a right-hand side graph as instances of the Graph class. Rules further contain node mappings between the LHS and the RHS which are omitted here for better readability. Graphs consist of a set of Nodes and a set of Edges. Nodes can additionally contain a set of Attributes. These three kinds of model elements are typed by their corresponding concepts in the Ecore meta model of EMF (depicted in blue). Rule inherits from Unit and therefore can contain a number of Parameters. A typical use of parameters is to pass an attribute value (e.g. a name) of a node to be matched to the rule. To constrain the applicability of a

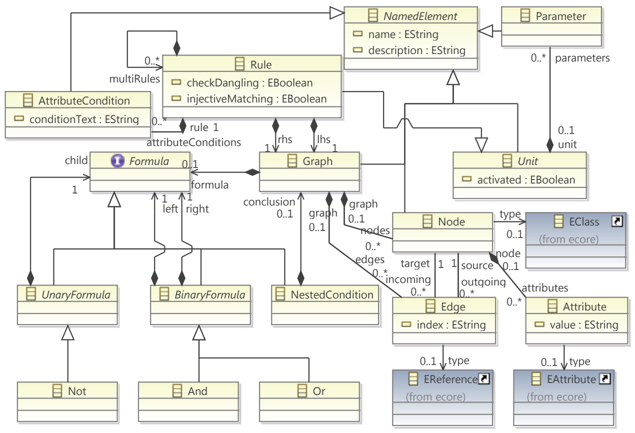


Fig. 1. Part of the Henshin meta model depicting details of rules.

rule, the meta model contains concepts for modeling nested graph conditions [5] as well as attribute conditions.

2.2 Graph pattern matching in Henshin

Henshin does neither impose any restrictions on the structure of the LHS, nor does it require to pass parameters that could be used as a starting point for the graph pattern search (as, e.g., required in Fujaba). This high expressive power comes with the price that in the worst case the graph pattern matching problem in Henshin is equivalent to the subgraph isomorphism problem.

The graph pattern matching engine of Henshin solves this task by translating it into an equivalent constraint-solving problem (CSP) and by employing several heuristics to achieve efficiency in almost all practical scenarios. Every node in the LHS is considered as a *variable* in the CSP. For each of these variables, a corresponding *domain slot* is created which is essentially the solution space for this variable, i.e., all nodes in the host graph which are possible matches for this node. The graph pattern matching is realized by a recursive algorithm which i) removes impossible solutions for a variable by imposing constraints on their domain slot, and ii) locking variables to specific solutions and thereby constructing matches.

The efficiency of the pattern matching primarily depends on a) how much the constraints reduce the solution space of variables and b) the order in which the recursive algorithm locks variables. The order of variables can in Henshin be either manually optimized or fully automatically using a heuristic which sorts the variables on-the-fly based on the current host graph (enabled by default). The used constraints in the CSP can be categorized into different groups. The most basic kind of constraints are *type constraints*. They reduce the solution space to objects which are instances of a given class (or a subclass of it). If there exists an edge $x \xrightarrow{e} y$ and the source variable x is locked already to a specific node X, the solution space of y is restricted to all e-images of X. This kind of constraints is referred to as *reference constraints* in Henshin. Similarly, so-called *containment constraints* reduce domain slots to containment children or container nodes (cf. *containment references* in EMF [9]). There exist more kinds of constraints, but we omit them here due to lack of space.

3 Bad smells and their detection

In the following, we present a set of bad smells which can negatively affect the performance of the matching of the LHS. Detecting an instance of such a bad small does not imply that it is a performance problem. It is merely an indicator that there might be a problem which can cause an unnecessary bad transformation execution performance. Furthermore, the presented detectors have false positives and false negatives due to the decision to specify the detection mainly using Henshin rules on Henshin rules. Due to space restrictions, we cannot show all detectors in the paper in detail. Instead, we refer the interested reader to the detection Henshin rule set and the supporting source code available at `www.cse.chalmers.se/~tichy/henshin_bad_smells.zip`

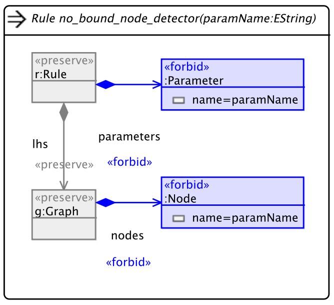
3.1 LHS not connected

Description: If a LHS is not a connected graph, it is not possible to use the edges between nodes to constrain possible matches of nodes. This means (in general) that two independent subgraphs must be matched.
Detector: The detector computes all (weakly) connected components of a LHS by a loop, which starts by marking an unmarked node of the LHS with a distinct marker and then iteratively marks with the same marker all nodes which are reachable by an edge from a marked node. This loop is executed until all nodes are marked. If the rule contains more than one marker object, the LHS is not connected.
Notes: The detector currently computes only (weakly) connected components. By supporting strongly connected components, the detection would also correctly identify a LHS where one of the two subgraphs of the LHS are only connected by an edge directed from one subgraph to the other. In this case, reference constraints can only be exploited in one direction of the rule.

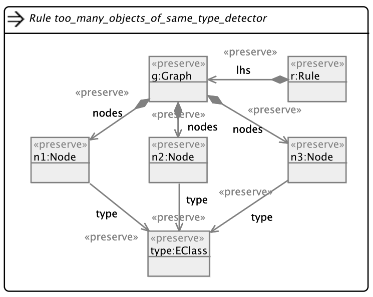
3.2 No bound node in the LHS

Description: If a node of the LHS is already known due to a rule parameter, matching of the other nodes of the LHS can start at a known point in the host graph. Hence, instead of searching the whole host graph for a matching only a local search must be performed. Consequently, the interpreter optimizes the matching order of the nodes by moving known objects to the front of the matching order.

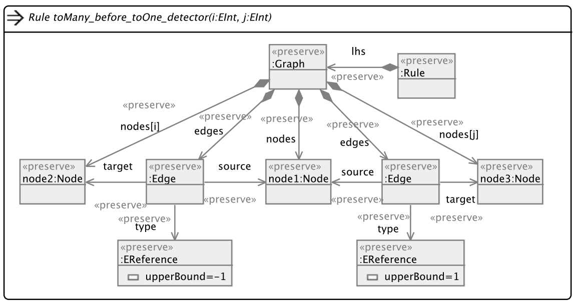
Detector: The detector checks whether a rule parameter exists with the same name as a node in the LHS. If the rule of a graph can be divided into several connected components, the detector also checks for each connected components since a matched node in one connected component does not affect (in general) the matching of the other connected components. Figure 2(a) shows the main detector rule. Additionally, the detector ignores multi-rules since they typically contain mappings to the kernel rule.
Notes: Note, the user of the transformation decides whether this parameter is used as input for the matching of the LHS or only as an output. In the latter case, the undetected bad smell would be a false negative.



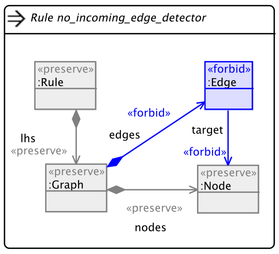
(a) Main part of the detector for rules without known node



(b) Detector rule for too many nodes of the same type



(c) Detector rule for binding a node over to-many edge before a node binding over to-one edge



(d) Main part of the detector rule for nodes without incoming reference

Fig. 2. Selected detectors

3.3 No bound node or node with attribute constraint in the LHS

Description: While the previous "no bound node" bad smell checks only whether a node in the LHS is known by a parameter, constraints on an attribute of the

node is another option to restrict the number of possible objects to match. The interpreter also places nodes with an attribute constraint as much as possible to the front of the matching order but behind known nodes.
Detector: The detector is an extension of the detector of the previous sections. It detects an LHS (or similarly to above a connected component of the LHS) which neither contains a known node nor a node with an attribute constraint. The reason to extend the previous detector and only report an LHS which additionally does not contain a known node is that a known node already improves the matching process by itself. Again, multi-rules are ignored since they contain mappings to the kernel rule.
Notes: A future option to increase the execution performance of Henshin transformation rules would be to support Map-based references as in Fujaba. This would improve the matching of nodes with attribute constraints when the attribute constraint is used as an unique identifier.

3.4 No incoming edges

Description: If a node has no incoming edge, all objects in the system of the same type are possible matches for this node since matches of other nodes cannot be used to restrict the number of potential objects for this node by reference constraints.
Detection: The major part of the detector for this bad smell is shown in Figure 2(d). Additionally, the detector checks that the node without incoming references does not already have a matching due to a parameter of the rule.

3.5 Preferring to-many over to-one references

Description: As described above, Henshin supports the optimization of the node matching order to improve the performance of the matching. If this option is not used the order of the matching is defined by the order of the nodes in the graph of the LHS.

In this case, we detect a bad smell if a node is matched which is reachable by a to-many reference before a node is matched which is reachable by a to-one reference. The to-one referenced node is faster to match as only one possible object can be matched whereas the to-many referenced node might have multiple possible matching objects.
Detector: Figure 2(c) shows the detector of this bad smell. In addition, it checks whether `node2` is matched before `node3`. For this, we exploit that the reference between `Graph` and `Node` is ordered and thus we can check the indices of the nodes in this reference using the newly implemented support for edge indices in Henshin.

3.6 Too many nodes of the same type

Description: Henshin employs type constraints to restrict the possible matching of nodes to those objects of the correct type. If a lot of nodes in the LHS are

typed over the same type, then the type constraints might lose their benefits. In the extreme case, if only one type is used, then type constraints do not restrict the matchings at all.
Detector: The detector checks whether the number of nodes with the same type exceeds a certain threshold. For our evaluation, we used a threshold of 3 nodes with the same type. Figure 2(b) shows the detector rule.

3.7 Using generic types

Description: If the typing used for the node in the LHS of a Henshin rule is too generic, the type constraints in the Henshin interpreter do not gain that much performance. An example would be the use of EObject, which is the supertype of all EMF generated classes, as type of a node.

Another case is the generic use of the Trace class provided by Henshin to specify tracing between objects in exogeneous model transformations. In order to improve the matching of the trace objects, it is beneficial to create and use different subclasses of Trace for each tracing information, such that type constraints on traces can be exploited in the interpreter.
Detector: The detector detects rules which include one node which is typed using EObject, Trace or an abstract class.

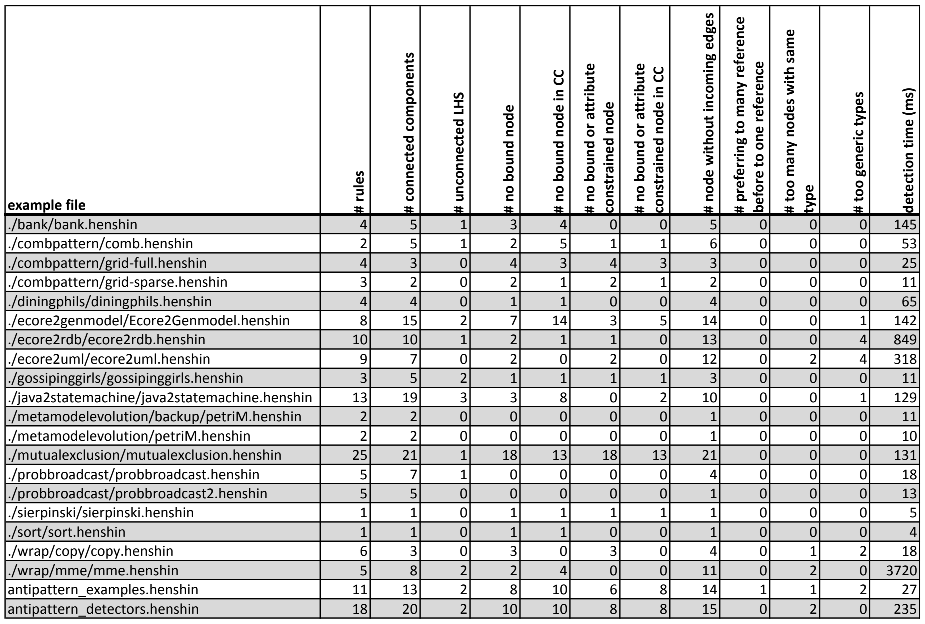
4 Evaluation

As an early evaluation, we executed the detectors on all examples delivered as part of Henshin, our own example file, and on the detector rules themselves. Figure 3 shows an overview of the results. The two first columns contain as reference the number of rules and connected components in each example file. Note that some rules do not include an LHS, i.e., they are always applicable. The number of connected components can be less than the number of rules in these cases.

Mostly, the bad smells “no bound node” and “node without incoming edges” have been detected in the evaluation examples. However, often these rules contain at least a node with an attribute constraint. Another aspect shown in the evaluation is that some of the examples are graph transformation benchmarks or generic examples like mutualexclusion.henshin and sierpinski.henshin where a number of performance bad smells are not surprising. The bad smell “preferring to many reference before to one reference” seems to be a known issue. Finally, when running the detectors on themselves, we are also guilty of some performance bad smells.

5 Related Work

There exists some previous work in the area of model transformations and performance. Varró et al. [15] present a performance benchmark for different graph



example file	# rules	# connected components	# unconnected LHS	# no bound node	# no bound node in CC	# no bound or attribute constrained node	# no bound or attribute constrained node in CC	# node without incoming edges	# preferring to many reference before to one reference	# too many nodes with same type	# too generic types	detection time (ms)
./bank/bank.henshin	4	5	1	3	4	0	0	5	0	0	0	145
./combpattern/comb.henshin	2	5	1	2	5	1	1	6	0	0	0	53
./combpattern/grid-full.henshin	4	3	0	4	3	4	3	3	0	0	0	25
./combpattern/grid-sparse.henshin	3	2	0	2	1	2	1	2	0	0	0	11
./diningphils/diningphils.henshin	4	4	0	1	1	0	0	4	0	0	0	65
./ecore2genmodel/Ecore2Genmodel.henshin	8	15	2	7	14	3	5	14	0	0	1	142
./ecore2rdb/ecore2rdb.henshin	10	10	1	2	1	1	0	13	0	0	4	849
./ecore2uml/ecore2uml.henshin	9	7	0	2	0	2	0	12	0	2	4	318
./gossipinggirls/gossipinggirls.henshin	3	5	2	1	1	1	1	3	0	0	0	11
./java2statemachine/java2statemachine.henshin	13	19	3	3	8	0	2	10	0	0	1	129
./metamodelevolution/backup/petriM.henshin	2	2	0	0	0	0	0	1	0	0	0	11
./metamodelevolution/petriM.henshin	2	2	0	0	0	0	0	1	0	0	0	10
./mutualexclusion/mutualexclusion.henshin	25	21	1	18	13	18	13	21	0	0	0	131
./probbroadcast/probbroadcast.henshin	5	7	1	0	0	0	0	4	0	0	0	18
./probbroadcast/probbroadcast2.henshin	5	5	0	0	0	0	0	1	0	0	0	13
./sierpinski/sierpinski.henshin	1	1	0	1	1	1	1	1	0	0	0	5
./sort/sort.henshin	1	1	0	1	1	0	0	1	0	0	0	4
./wrap/copy/copy.henshin	6	3	0	3	0	3	0	4	0	1	2	18
./wrap/mme/mme.henshin	5	8	2	2	4	0	0	11	0	2	0	3720
antipattern_examples.henshin	11	13	2	8	10	6	8	14	1	1	2	27
antipattern_detectors.henshin	18	20	2	10	10	8	8	15	0	2	0	235

Fig. 3. Detected bad smells in Henshin example rule sets

transformation based model transformation tools. Another paper which discusses model transformations and performance is [2] which compares different styles of model transformations with respect to their performance in different benchmarks. Mészáros et al. present how to optimize the performance of model transformations manually by exploiting knowledge of the designer of the transformation system and automatically by an application specific generated matching algorithm [8]. However, none of these papers considers the peformance impacts of bad smells.

Another line of research is the definition of metrics for model transformations [17, 13]. These papers' approaches mainly adapt metrics from programming languages to model transformations. While [13] is mainly concerned with maintainability metrics, a few of the metrics presented in [17] are related to performance. Similarly, the catalog of model transformation refactorings presented in [18] also includes a few refactorings which improve the performance of the model transformations in the experimental evaluation.

Specifically for Henshin graph transformations, Taenzter et al. also define bad smells and refactorings [11]. However, they do not address performance. In summary, there exists no work which presents bad smells or anti patterns specifically for the performance of model transformations.

6 Conclusion and Future Work

We presented performance bad smells in Henshin model transformation rules and corresponding detectors which were also specified as Henshin model transformations. The current version of the detection suffers from the design decision to specify the detectors also mostly with Henshin rules. Consequently, only an analysis on a syntactical level is possible which leads to false positives and false negatives.

Future evaluations of the bad smells will compare the performance of the presented detector rules with refactored rules which have less performance bad smells. Early investigations show that refactoring of rules to remove the bad smells is not a simple change but might require more extensive changes in several rules and how the rules interact. It remains to be seen whether these refactorings have a positive effect on performance but a negative effect on other qualities like readability.

We will investigate an extension of the Henshin interpreter engine in order to assess the impact of the identified bad smells at runtime. For example, we can measure and plot the number of possible matches for a node during the matching of the LHS in order to identify whether the bad smell "no bound node in LHS" for the analysed rule is really problematic or not.

Furthermore, the presented bad smells might be applicable and extended for other model transformation languages as well.

Acknowledgements

The research leading to these results has received partial funding from the European Union's Seventh Framework Program (FP7/2007-2013) for CRYSTAL – Critical System Engineering Acceleration Joint Undertaking under grant agreement No 332830 and from Vinnova under DIARIENR 2012-04304. This work was also done in collaboration with the project ENSURE which is funded by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

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 D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010.
2. R. Bruni and A. Lluch-Lafuente. Evaluating the performance of model transformation styles in maude. In F. Arbab and P. C. Ölveczky, editors, *FACS*, volume 7253 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2011.
3. A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. JTL: A bidirectional and change propagating transformation language. In *Software Language Engineering - Third International Conference, SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 183–202. Springer, 2010.

4. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.
5. A. Habel and K.-h. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical. Structures in Comp. Sci.*, 19(2):245–296, Apr. 2009.
6. F. Hermann, S. Gottmann, N. Nachtigall, B. Braatz, G. Morelli, A. Pierre, and T. Engel. On an automated translation of satellite procedures using triple graph grammars. In *Theory and Practice of Model Transformations*, volume 7909 of *Lecture Notes in Computer Science*, pages 50–51. Springer Berlin Heidelberg, 2013.
7. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.
8. T. Mészáros, G. Mezei, T. Levendovszky, and M. Asztalos. Manual and automated performance optimization of model transformation systems. *STTT*, 12(3-4):231–243, 2010.
9. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2. edition, 2009.
10. G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2004.
11. G. Taentzer, T. Arendt, C. Ermel, and R. Heckel. Towards refactoring of rule-based, in-place model transformation systems. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, AMT '12, pages 41–46, New York, NY, USA, 2012. ACM.
12. L. Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–124, 2008.
13. M. van Amstel, M. van den Brand, and P. Nguyen. Metrics in model transformations. In S. Ducasse, L. Duchien, and L. Seinturier, editors, *BENEVOL 2010 (9th Belgian-Netherlands Software Evolution Seminar, Lille, France, December 16, 2010. Proceedings of Short Papers)*, pages 1–5. Université Lille 1, 2010.
14. G. Varró, F. Deckwerth, M. Wieber, and A. Schürr. An algorithm for generating model-sensitive search plans for emf models. In Z. Hu and J. de Lara, editors, *ICMT*, volume 7307 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2012.
15. G. Varro, A. Schurr, and D. Varro. Benchmarking for graph transformation. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '05, pages 79–88, Washington, DC, USA, 2005. IEEE Computer Society.
16. D. Varró and A. Balogh. The model transformation language of the {VIATRA2} framework. *Science of Computer Programming*, 68(3):214 – 234, 2007.
17. A. Vignaga. Metrics for measuring atl model transformations. Technical Report TR/DCC-2009-6, 2009.
18. M. Wimmer, S. Martínez, F. Jouault, and J. Cabot. A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, 11(2):2:1–40, Aug. 2012.
19. A. Zündorf. Graph pattern matching in PROGRES. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *TAGT*, volume 1073 of *Lecture Notes in Computer Science*, pages 454–468. Springer, 1994.