Monitoring the Execution of Declarative Model Transformations

Raffaela Groner, Sophie Gylstorff, Matthias Tichy
raffaela.groner@uni-ulm.de, sophie.gylstorff@uni-ulm.de, matthias.tichy@uni-ulm.de
Ulm University, Ulm, Germany

Abstract

Model transformations, applied at design and run time, are key artifacts in Model-Driven Software Engineering. The monitoring of a transformation's execution is a prerequisite to enable a software engineer to identify performance bottlenecks and improve transformations. Monitoring is particularly relevant for declarative model transformations since the order of execution is not explicitly defined but instead the result of internal heuristics of the transformation engine. In this paper, we present how we monitor the execution of Henshin model transformations using Kieker as well as the resulting monitoring overhead. We show that the monitoring overhead depends on the size of the input model and that it is between 17.03% and 28.44%.

1 Introduction

Model-Driven Software Engineering (MDSE) aims at handling the continuous growth in the complexity of software by using models as key artifacts. An important operation on these models is a model transformation, which translates an input model into an output model.

Performance is a very important property of model transformations as models depicting productive environments can get quite large. For example, a model from an industrial partner, that describes an electronic control unit of a car, consists of over 170000 model elements, and the application of transformations may take several hours.

Model transformations are usually interpreted by a transformation engine whose internal optimizations and their effects on the performance are unknown to the software engineer. This is particularly relevant for declarative model transformations as, in contrast to operational model transformations, the order of execution is not specified by the software engineer but is instead the result of unknown heuristics. Hence, if the performance is not satisfactory, the engineer lacks important knowledge to identify performance bottlenecks and, thus, is not able to change the transformation in order to improve its performance.

Monitoring of model transformations is so far restricted to simply measuring the overall execution time [3]. This information help to identify performance bottlenecks, but the reasons why a specific transformation takes long aren't given. Some research focuses on the optimization of the engines heuristic, e.g. [2], whereby a transformation engineer doesn't get any further information how these optimizations influence her transformation during its execution. To our knowledge there do no exist any approaches supporting detailed monitoring of the execution of declarative model transformations including the results of the engine's heuristics. We need something similar to the approach by Debray [1] which monitors the execution of Prolog programs, e.g, monitoring when and where backtracking occurs.

In this paper, we present how we monitor Henshin model transformations. Henshin [5] is a declarative model transformation language based on the graph transformation paradigm. That means, the model transformation consists of a precondition, the so-called left-hand side (LHS), specified as a graph, and a postcondition, the so-called right-hand side (RHS), also specified as a graph. The transformation is executed by finding a subgraph, isomorphic to the LHS, in the model and changing that subgraph such that it becomes isomorphic to the RHS.

Specifically, our monitoring approach for Henshin answers the following questions:

Q1 How do we receive the order in which the elements of the LHS are chosen to find an isomorphic node in the input model?

Q2 How do we get the number of model elements examined for each element in the LHS?

Q3 How can we monitor how binding decisions of a model element to an element of the LHS affect candidate sets for other LHS elements?

Q4 How can we monitor where and when backtracking occurs?

Q5 How can we measure how long the transformation execution takes?

In the following Section 2, we present how we use Kieker [4] to monitor the execution and answer the above's questions. We furthermore measured the monitoring overhead. We conclude in Section 3 and give an overview of future work.

2 Monitoring of Transformations

We extended the Henshin Eclipse Plugin to monitor the execution of model transformations with the help

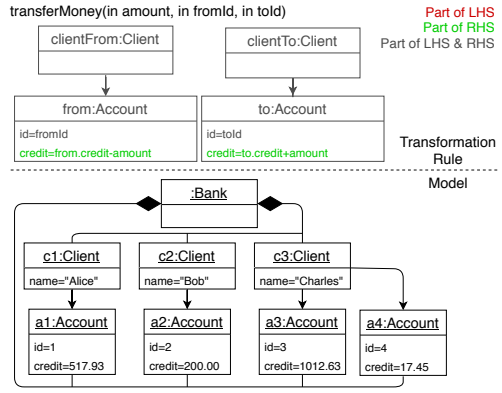

Figure 1: The transformation rule *transferMoney* and the used input model

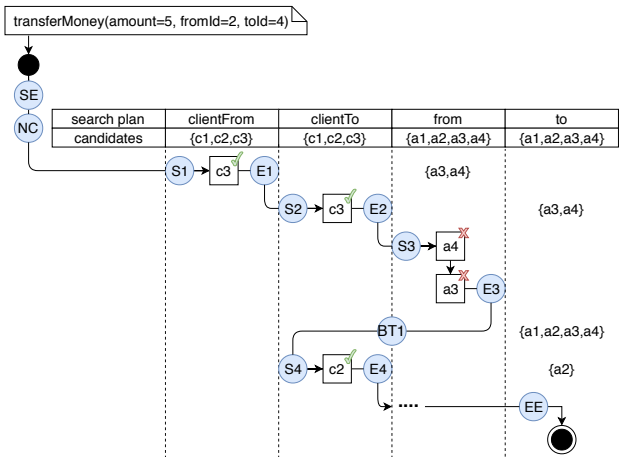

Figure 2: Part of the execution trace from *transferMoney*

of Kieker [4]. To illustrate which data we collect we use the example rule *transferMoney* that is shown in Figure 1. In this figure, the LHS and RHS are combined. This transformation rule defines the transfer of a given amount of money, defined by *amount*, between two accounts. The amount of money is subtracted from an account with the given *id fromId*, and added to an account with the given *id toId*. Each of these accounts is referenced by a client that owns it. The values for the parameters *amount*, *fromId* and *toId* are set before application.

To illustrate how a transformation is executed and which data is monitored we describe how the engine executes the transformation *transferMoney*. In this example, we apply *transferMoney* with *amount*=5, *fromId*=2 and *toId*=4 on the input model shown at the bottom of Figure 1. Figure 2 illustrates a part of the execution trace from *transferMoney* where the circles mark measuring points. The table at the top shows the order in which the nodes of the LHS are investigated in the row *search plan* and the possible model elements for each node in row *candidates*.

By the time the execution starts, we measure the start time in measuring point *SE*. For each node of the LHS we monitor after the execution starts in measuring point *NC* the number of candidates. We monitor the number of candidates, because in worst case their cross product is the maximum number of model elements which need to be investigated. This measurement is also the answer to **Q1**, because at this point in time the nodes are already ordered and so we receive the search order of the nodes implicit by their order in *NC*. Then the engine starts to find a model element for the node *clientFrom*. Here, we monitor in measuring point *S1* the start time of this search and the number of possible candidates. The first model element that is investigated is *c3*, depicted in Figure 2 as a rectangle, which is a match for *clientFrom*. Since *transferMoney* defines that *clientFrom* needs to be the owner of *from*, the candidates for *from* are reduced to $\{a3,a4\}$. This reduction is depicted in Figure 2 in the column *from*. With this, the search for a match for *clientFrom* is finished and we measure in measurement point *E1* the number of model elements which were investigated until a match was found, here 1. We also measure that the number of possible candidates for *from* is reduced to 2 model elements and the end time. The data from measurement point *S1* and *E1* help to understand how long it takes until a match for *clientFrom* was found, how many candidates were available, how many candidates are investigated and how the found match influences the candidates for *from*. So the measuring points *S* and *E* are the answers for the questions **Q2** and **Q3**.

Then, just as with *clientFrom*, the engine looks for a match for *clientTo* and for *from*. For *from* both candidates $\{a3,a4\}$ are investigated, but none is a match for *from* because *from* defines the restriction that an account with *id*=*fromId*=2 has to be found. Since there is no candidate left for *from* the search for its match is finished. In such a case that no candidate is a match for a node the engine uses backtracking. This means the engine revokes the found match for the previous node and tries to find a new one for it. So the match for *clientTo* is revoked and thus also its restriction of the possible candidates for *to*. As answer for **Q4** we use the measuring point *BT1*, in which we monitor that the search goes back to *clientTo* and the point in time before its match and the resulting restrictions are revoked. That data provides the information that no match was found for *from* and that it was also required to revoke the match for *clientTo*. Then again, a match is searched for *clientTo*. This matching process continues until a match was found for each node and the transformation rule can be applied, or until there are no more possible candidates and the execution of the transformation rule aborts. In both cases we measure in measurement point *EE* the point in time the execution of the transformation ends. So the answers for **Q5** are the measuring points *SE* and *EE*.

With the help of this monitoring an engineer gains three important pieces of information about the execution of *transferMoney*: (1) The duration of the transformation execution. (2) The engine needs to backtrack *clientTo* because no match for *from* was found. (3) The candidates for *from* are restricted by *clientFrom*, for which the backtracking of *clientTo* does not help to find a match for *from*. For this example the engineer only needs to set the appropriate engine parameter and the engine swaps the order of *from* and *clientTo* in the search plan and the amount of backtracking for *clientTo* and the number of investigated candidates for *from* will be reduced. In this small example this swap will decrease the average execution duration with monitoring from 7.22ms to 6.96ms[1].

To investigate the monitoring overhead we applied *transferMoney* with and without monitoring each 100 times[1]. We randomized for each application the values of *fromId* and *toId*, and measured the execution duration of each application with and without monitoring. This measurement is repeated on 11 input models, where each has a different number of *Accounts*. Table 1 shows the average of the measured durations and the variance with ($\varnothing t_M$, σ_M^2) and without ($\varnothing t$, σ^2) monitoring. Table 2 shows the average overhead for the measuring points in percent.

Number of Accounts	$\varnothing t$ in ms	$\varnothing t_M$ in ms	σ^2	σ_M^2	Overhead
160	6.982	8.830	0.137	2.474	26.46%
240	7.164	8.556	0.019	2.119	19.44%
320	7.252	9.314	0.037	2.599	28.44%
480	7.413	9.289	0.153	1.818	25.31%
640	7.584	9.474	0.123	1.517	24.93%
960	7.851	9.376	0.229	2.611	19.42%
1280	8.133	9.639	0.376	2.495	18.51%
1920	8.752	10.314	0.676	2.991	17.85%
2560	9.378	10.975	1.389	4.271	17.03%
3840	10.556	12.399	2.313	4.803	17.46%
5120	11.793	13.802	4.605	4.097	17.03%

Table 1: Average durations of *transferMoney*

It should be considered that the measured values in table 1 and 2 aren't very accurate. These examples are so small, that we had to measure at the nanosecond range. So also external factors like I/O or caching had a major influence on the run time. Another negative influence on our monitoring overhead is that we use the AsciiFileWriter from Kieker which is slower than e.g. the BinaryFileWriter.

3 Conclusion

Monitoring provides the foundation to understand the trace of execution and the performance of a declarative model transformation. We presented our monitoring implementation for Henshin.We describe where and which data we collect to receive the execution duration and to trace how the input model is searched for subgraphs, like how many candidates are investigated, how a match influences candidates for other matches or how often a match needs to be revoked. Those information can help a transformation engineer to understand why her transformation takes so long. Also we showed that the monitoring overhead depends on the measuring points and the size of the input model. Currently our monitoring implementation takes only the search process to find a isomorphism of a subgraph in the input model and the whole execution duration into account. So we will extend it by adding new measurement points e.g. to measure the time it takes to load the input model and to monitor the execution of control structures like a loop.

Acknowledgements This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Ti 803/4-1.

Number of Accounts	*SE & EE*	*NC*	*S & E*	*BT*
160	2.26%	0.36%	1.71%	2.85%
240	1.51%	0.64%	1.86%	2.04%
320	3.30%	1.60%	2.18%	3.91%
480	2.71%	0.82%	1.19%	3.26%
640	3.05%	1.08%	1.98%	3.23%
960	3.09%	1.24%	1.36%	3.48%
1280	3.02%	0.83%	1.32%	3.65%
1920	3.18%	1.26%	1.12%	3.64%
2560	2.94%	1.14%	1.07%	4.32%
3840	3.21%	1.84%	0.82%	4.45%
5120	3.54%	1.50%	0.38%	4.27%

Table 2: Monitoring overhead for measuring points

References

[1] S. K. Debray. "Profiling prolog programs". In: *Software: Practice and Experience* 18.9 (1988), pp. 821–839.

[2] G. V. Batz, M. Kroll, and R. Geiß. "A first experimental evaluation of search plan driven graph pattern matching". In: *International Symposium on Applications of Graph Transformations with Industrial Relevance*. Springer. 2007, pp. 471–486.

[3] W. Piers. "ATL 3.1–Industrialization improvements". In: *Proceedings of the 2nd International Workshop on Model Transformation with ATL*. Citeseer. 2010.

[4] Kieker Project. *Kieker web site*. http://kieker-monitoring.net/ Visit: 08.06.2018. 2013.

[5] D. Strüber et al. "Henshin: A usability-focused framework for emf model transformation development". In: *International Conference on Graph Transformation*. Springer. 2017, pp. 196–208.

[1] Ubuntu 18.04, 2 Pentium Dual-Core E5200 at 2.50GHz