Towards Debugging the Matching of Henshin Model Transformations Rules

Matthias Tichy, Luis Beaucamp, and Stefan Kögel
Institute of Software Engineering and Programing Languages, Ulm University, Germany
Email: <firstname.lastname>@uni-ulm.de

***Abstract*—Model Driven Engineering aims at improving effectiveness and efficiency of software engineering. Model transformations are a key artifact in model driven engineering as they enable transforming models into other artifacts for further processing, e.g. analysis models or executable code. However, particularly graph transformation based model transformation tools often lack detailed debugging capabilities. In this paper, we sketch how to debug the matching step of the execution of Henshin model transformation rules.**

I. Introduction

Software engineering with models often requires the application of model transformations, e.g., chains of model transformations to translate state machines into code or operations in the editor to change the models. Hence, model transformations are a key element of model-driven software engineering.

Existing research has identified the ability to analyze models as a key driver w.r.t. to successful application of model-driven engineering in practice [1]. Debugging is a specific analysis technique – not only with respect to finding defects but also to understanding how the model transformation works, e.g. for adding functionality. Debugging Henshin rules, a graph-transformation based model transformation tool, has the challenge that its execution process does not follow the typical sequential order of imperative languages where one usually executes statement after statement.

Several model transformation frameworks offer debugging support that offers at least basic break points and stepwise execution on the transformation rule level. Some frameworks extend this support by offering undo/redo (e.g. QVT-o [2]), conditional breakpoints (e.g. QVT-o, VIATRA [3]), visualization of matchings (e.g. GReAT [4], VMTS [5]), or even the modification of the current match during debugging [5]. All of the above debugging features work on the rule level, i.e., they step over the matching phase of a rule. To the best of our knowledge, there is no graph transformation approach that enables developers to debug the matching step of a rule.

The contribution of this paper is a sketch how the matching step can be debugged, which is the most important part of the execution of single Henshin transformation rules.

II. Henshin Transformation Language

Henshin supports single rules, containing a left hand side (LHS) describing the precondition of the rule and a right hand side (RHS) describing the postcondition of the rule. Informally, a Henshin transformation rule is executed by first finding a match of the left hand side on the model resp. host graph (typically a subgraph isomorphism) and then changing the match such that it is a match of the right hand side.

We illustrate the techniques presented in this paper using the `transferMoney` rule from the Bank Example [6] provided with Henshin (cf. Figure 1). It represents the possibility to transfer some amount of money from one account to another while ensuring that the sender's credit is sufficient and that the sender cannot transfer money to himself from the receiver's account by entering a negative amount.

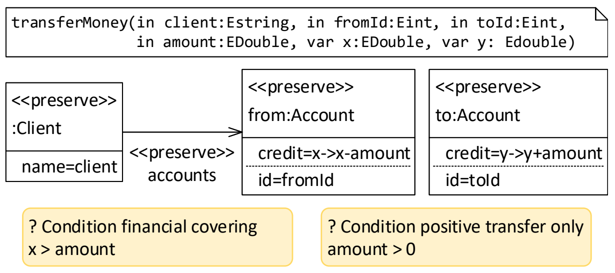
transferMoney(in client:Estring, in fromId:Eint, in toId:Eint,
in amount:EDouble, var x:EDouble, var y: Edouble)
<<preserve>>
:Client
name=client
<<preserve>>
accounts
<<preserve>>
from:Account
credit=x->x-amount
id=fromId
<<preserve>>
to:Account
credit=y->y+amount
id=toId
? Condition financial covering
x > amount
? Condition positive transfer only
amount > 0

Fig. 1. Bank example - transformation rule

Henshin uses a constraint based approach for finding matches of the left hand side in the host graph [7]. In a first step, a search plan is derived containing a *variable* for every node in the left hand side. For each variable, a corresponding *domain slot* is created which contains a set of all potential candidates for this node – initially all instances of the node's type. Furthermore, the search plan also defines which constraints should be enforced, e.g., constraints on attributes as well as on references between nodes.

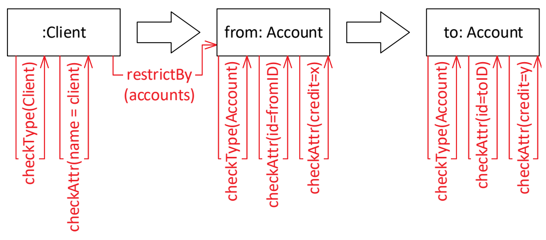

Fig. 2. Bank example - search plan

Figure 2 shows the search plan of the Henshin rule `transferMoney` of the bank example. The order of matching the variables is shown by the rectangles and the big arrows.

Furthermore, for each variable the constraints are annotated. For example, for the `:Client` variable first the type is checked and then the attribute constraint is checked. Finally, the set of candidates for the `from:Account` is restricted to those which are reachable via the `accounts` reference from the current candidate for the `:Client` variable.

The match of the left hand side is then computed by a recursive back-tracking algorithm which binds potential candidates, i.e., objects in the model (the model provided with the Henshin example contains the clients `Alice`, `Bob`, `Charles`), to variables while removing impossible candidates from the domain slots by imposing the constraints derived from the left hand side.

III. Debug Steps for Henshin Rules

Our approach follows the usual debugging steps (step into, step over, step return, run until) found in all typical debugging environments. As described above the Henshin interpreter uses a recursive backtracking algorithm to find a match for the left hand side of the rule. Each recursive call covers the matching of a single variable.

The matching of a single variable consists of several activities of the matching process as indicated in the search plan (cf. Figure 2): selection of a candidate as well as checking/enforcing the different types of constraints. Multiple instances of each constraint type are possible, e.g., multiple attribute constraints. Each of these activities is similar to a function in an imperative program that "call" each other, e.g., the function responsible for selection of candidates calls the different functions for checking/enforcing constraints, and the function responsible for checking/enforcing all attribute constraints calls a function for checking an individual attribute. Hence, a specific state during matching in our running example has a "call stack" (cf. Table I).

TABLE I
Call stack for a specific state during debugging.

	activity	example
4.	*Constraint Instance*	check attribute constraint: name == "Bob"
3.	*Constraint Type*	check all attribute constraints
2.	*Candidate*	check the candidate bob
1.	*Variable*	select candidate for the variable :Client

The search plan (cf. Figure 2) shows that the variables are matched in the order `:Client`, `from:Account`, `to:Account`. Hence, the first debugging state is matching the variable `:Client`. With step over, we would complete the matching of the variable, i.e., selecting a candidate which satisfies all constraints, which in our running example is `Bob`. However, if we step into, the debugging state changes to selecting the candidate `Charles` as it is the first object in the model. This step into allows us to iterate with subsequent step overs over all candidates for the variable.

But, if we further step into, we can iterate over constraint type and instances of the constraint type. In our running example, we first check the type (`Client` in our example) and then attribute constraints like `name == ''Bob''`. Unfortunately, the current candidate has the name `''Charles''`. Hence, the attribute constraint is not fulfilled and stepping further leads to considering the next candidate (`Bob`). Here, the type and attribute constraints are satisfied. Executing the reference constraint ensures that the set of candidates for the account variable `from:Account` is restricted to the candidate `2:Account` as this is the only account `Bob` has. However, we see later that this account does not have enough credit. The final candidate for the `:Client` variable also does not satisfy all constraints.

While step into covers all states, step over usually stays on the same "call stack" level, e.g., iterating over all candidates for a variable or iterating over all constraint types. Step return simply executes all other activities until a higher level is reached, e.g. if the current debug state is investigating the candidate `Charles`, step return executes all steps until the matching process of the variable `from:Account` starts.

Finally, run until executes all steps until a certain activity takes place, e.g., until a certain candidate is investigated, a certain constraint is checked, or a matching of a certain variable starts.

IV. Conclusion and Future Work

Currently, we have implemented the debugging steps outlined in Section III in Henshin and refactored the interpreter in order to enable step-wise execution of the matching process. We are currently working on integrating the different step actions into the standard Eclipse debugging environment and are developing specific visualizations for the variable view as well as the call stack. Whether and how much our approach actually improve efficiency and effectiveness will be a focus of future user studies.

Acknowledgment This work was funded by the German Research Foundation (DFG) as part of the DFG Priority Programme 1593 (TI 803/2-2 and TI 803/4-1).

References

[1] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Software & Systems Modeling*, pp. 1–23, 2016.

[2] M. Wimmer *et al.*, "Reviving QVT relations: model-based debugging using colored petri nets," *Model driven engineering languages and systems*, pp. 727–732, 2009.

[3] "Advanced features of the viatra framework," http://static.incquerylabs.com/projects/viatra/viatra-docs/ViatraAdvanced.html, (Accessed on 07/14/2017).

[4] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai, "The graph rewriting and transformation language: Great," *Electronic Communications of the EASST*, vol. 1, 2007.

[5] T. Mészáros, P. Fehér, and L. Lengyel, "Visual debugging support for graph rewriting-based model transformations," in *EUROCON, 2013 IEEE*. IEEE, 2013, pp. 482–488.

[6] C. Krause, "Henshin bank example," https://www.eclipse.org/henshin/examples.php?example=bank.

[7] M. Tichy, C. Krause, and G. Liebel, "Detecting performance bad smells for henshin model transformations," in *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, 2013.