A Proposal of Features to Support Analysis and Debugging of Declarative Model Transformations with Graphical Syntax by Embedded Visualizations

Florian Ege
Inst. of Software Engineering and Programming Languages
Ulm University
florian.ege@uni-ulm.de

Matthias Tichy
Inst. of Software Engineering and Programming Languages
Ulm University
matthias.tichy@uni-ulm.de

***Abstract*—In model-driven software engineering (MDSE), chains of model transformations are used to turn a source model via a series of intermediate models into a target artifact. At times such a transformation chain does not deliver the expected result, either because a particular transformation step fails due to unmet preconditions, or the produced target artifact is not the desired one. To better understand the transformation process, and to locate and correct defects in the models or transformations involved, developers need appropriate tool support for analysis and debugging.**

MDSE tools provide a spectrum of techniques for analysis. These range from model checking approaches for proving logical properties of transformations to low-level stepwise debugging functionality that exposes how particular algorithms, e.g., graph matching, are implemented. However, these existing analysis features often do not present concrete suggestions directed at locating and fixing defects, or require developers to reason about their models and transformations in a procedural way.

We focus on declarative model-to-model transformations with graphical syntax and consider defects located in source models or transformation specifications. For each of those defects, we sketch how a specific approach based on visualizing information integrated in the graphical syntax could support identifying and fixing that defect. These techniques aim towards enabling developers to analyze models and transformations on the same level of abstraction and with representations in the same syntax they normally work with.

***Keywords*-declarative model transformations, graphical syntax, analysis, debugging**

I. Introduction

Model-driven software engineering (MDSE) aims at dealing with complexity by providing a higher level of abstraction for designing software systems. In MDSE, developers create domain-specific high-level models. Intermediate artifacts, like lower-level models or textual source code are then automatically derived from high-level models by applying model transformations. Model transformations are expressed in transformation languages that follow particular programming paradigms and have their own specific textual or graphical syntax (cf. [1]).

Declarative, endogeneous model transformations, as implemented by Henshin [2], are specified by defining a set of model elements that should be matched against a part of the source model and some modifications related to those elements. The target model is then constructed by applying the modifications on the matched part of the source model, whereby elements can be preserved, removed, new ones created, or their attributes be modified.

At the level of abstraction, at which developers work, they see a declarative transformation as an atomic modification of a part of the source model to create the target model. The declarative definition doesn't specify a detailed operational semantics, i.e., a specific execution order of basic transformation steps (matching, creation, removal or modification of model elements). Declarative definitions express a transformation by specifying how the target model is structured in contrast to how exactly it is created procedurally. Developers don't need to care about those details, while it is left to the underlying transformation engine to perform this efficiently. This constitutes the main advantage of declarative specifications.

However, on the other hand this abstract view makes it difficult to debug faulty source models or transformations (cf. [3]), that can cause target artifacts to be incorrect or invalid according to the terminology introduced by Hibberd et al. in [4]. Although most developers are familiar with debugging techniques for imperative programming languages, like stepping through statements, these are not directly applicable to declarative transformations. Imperative languages are based on an execution model that transforms program state by a sequence of steps. This fits the implementation of algorithms in the transformation engine, but not the declarative atomic view on transformations that developers have.

In this paper, we focus on declarative transformations with graphical syntax. We discuss some problems that we experience regularly when working with model transformations, e.g., why a transformation is not applicable to a model, and propose features that use visualizations embedded in the graphical syntax to aid in analysis and debugging. To illustrate our approach, we consider a simple use case of endogeneous transformations as a running example. After looking at related work in Sec. II, we introduce our running example in Sec. III. In Sec. IV, we then propose features to support analysis and debugging at the level of abstraction of models and transformation artifacts by sketching how visualizing information embedded in the graphical syntax of models and transformation specifications can be used to suggest, e.g., how an artifact could be modified to make a transformation applicable. We conclude by giving a summary of this paper and outlining future work in Sec. V.

II. Related Work

In this section we give a brief overview of related work, focusing on techniques to debug model transformations or prove properties of them.

For the Henshin tool, a debugging approach for the matching phase of a model transformation has been presented in [5]. This debugger allows stepping through the matching attempts of the engine in sequence. The manual effort is reduced by features like, e.g., letting the algorithm run until a breakpoint condition is reached.

For transformation languages based on the graph rewriting paradigm, Mészáros et al. [6] present techniques for the visual animation of ongoing transformations by showing pairings of elements from the rule-LHS and the source graph. The execution of the transformation can then be observed stepwise, with mechanisms like breakpoints, that are known from debugging imperative programs. The general purpose graph transformation tool Groove [7] supports debugging of transformations by remembering the sequence of applied transformation operations. This allows rolling back to a previous state in a chain of transformations, but makes the temporal order of steps very explicit. This is of course more natural, if an imperative algorithm is expressed by graph transformations in contrast to a declarative transformation of a model.

The above approaches are useful to see what is going on inside the transformation engine, e.g., during rule matching, yet this means working on a lower, procedural level when debugging transformations. Depending on the size of the models, a lot of (eventually failing) matches have to be inspected, with the downside that they are presented in an order depending on engine internals, instead of showing them in a meaningful order, like from most promising partial match to least. This can quickly become quite laborious.

Schönböck et al. [8] introduced transformation nets, a formalism based on typed and colored Petri nets, to represent the detailed steps taken by an engine during execution of a transformation, from initial matching of source model elements to creation of target model elements. This however also adopts a rather procedural view of a transformation. Moreover, the complex and expressive visualization of the transition nets is significantly distinct from the graphical syntax in which transformation rules or source models are represented. This makes it necessary to mentally connect different formalisms and translate between them.

With formal verification techniques, properties of models in connection with transformations can be proven. Schilling [9] introduces an approach for automatic verification of models that are represented by graphs. To keep the state space manageable, the model checker works on small graph patterns and proves properties of graph transformations inductively on small parts of the graph. However, if a transformation is not applicable at all to a model, a failure to verify that a correct model state can be derived by the transformation does not provide any insight about the defect in the transformation or the model that prevents the rule's application.

The Groove tool also allows for the appliction of model checking methods on systems, whose states can be mapped to associated graphs (cf. [10]). Properties to check are expressed with graph-based temporal logic formulas that are considered over a chain of transitions on graphs. This does however not provide concrete suggestions to fix the system in case of a negative proof of some property.

In summary, existing approaches to debugging model transformations either tend towards an imperative, procedural view on transformations, which is a mismatch to the declarative paradigm, or make statements about properties of a transformation (e.g., non-applicability) without providing detailed information as to the concrete cause and what artifact could be changed and how to alleviate the problem.

III. Parsing a Formal Language as Example

As our running example, we introduce a simple language for ordering pizza by the following grammar in Backus-Naur form, for which parse trees shall be constructed. The purpose of this example is not to claim that model transformations are a suitable technology for parsing tasks, but merely to serve as a commonly known domain for illustrating our proposed analysis features.

⟨*PizzaOrder*⟩ ::= ⟨*PizzaSpec*⟩ ①
| ⟨*PizzaSpec*⟩ **and** ⟨*PizzaOrder*⟩ ②

⟨*PizzaSpec*⟩ ::= **Pizza** ⟨*Name*⟩ ③
| **Pizza with** ⟨*Toppings*⟩ ④
| **Pizza** ⟨*Name*⟩ **without** ⟨*Toppings*⟩ ⑤
| **Pizza** ⟨*Name*⟩ **with** ⟨*Toppings*⟩ ⑥
| **Pizza** ⟨*Name*⟩ **without** ⟨*Toppings*⟩ **with** ⟨*Toppings*⟩ ⑦

⟨*Toppings*⟩ ::= ⟨*Topping*⟩ ⑧
| ⟨*Topping*⟩ **,** ⟨*Toppings*⟩ ⑨

⟨*Name*⟩ ::= **Capricciosa** | **Margherita** | **Rustica** | **...** ⑩

⟨*Topping*⟩ ::= **Artichokes** | **Garlic** | **Olives** | **Peperoni** | **...** ⑪

This grammar derives sentences like the following:

- *Pizza with Mushrooms, Onions, Parmigiano*
- *Pizza Contadina without Asparagus*
- *Pizza Funghi without Parsley, Olives with Ruccola and Pizza Hawaii without Ananas with Mozzarella, Oregano*

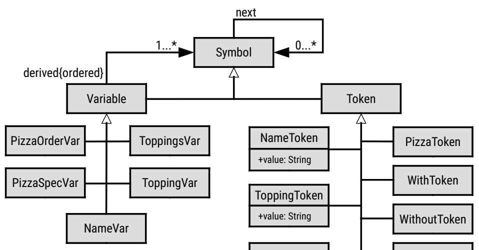

Figure 1. Metamodel for parse trees according to the pizza grammar.

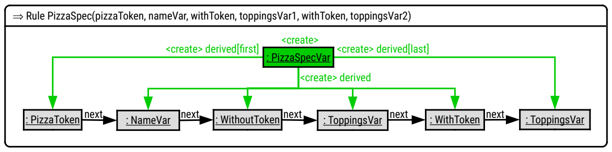

Figure 2. Transformation rule for the *PizzaSpec* grammar production no. 7. The grey elements are matched and preserved, the green ones are created by the transformation.

Fig. 1 depicts the metamodel containing classes for token and variable types. For each numbered production rule of the grammar, a corresponding graph transformation rule exists. Fig. 2 shows an example using a graphical syntax like that of Henshin. Those rules realize bottom-up construction of a parse tree by repeatedly applying the transformations that construct increasingly higher-up variable nodes, up to the root. Each symbol has `next` relations to its successors at different height levels of the tree. These connections are lifted up to variables in the parse tree by the transformation rules in Fig. 3.

Fig. 4 shows the parse tree for deriving the token sequence *"Pizza Margherita with Peperoni, Olives"*. The original source model consisted of the grey token objects on the bottom line and the `next` relations that string them together. On top of that, the transformation rules built up a parse tree with variable nodes, derivation relations and lifted `next` relations, all in green.

IV. Analysis and Debugging Scenarios in Model Transformations and Associated Visualizations

In this section, we discuss some common scenarios that are often encountered when developers need to debug model transformations. For each scenario, we outline its challenges and propose features to address them, using visualizations embedded in the graphical syntax for transformation rules

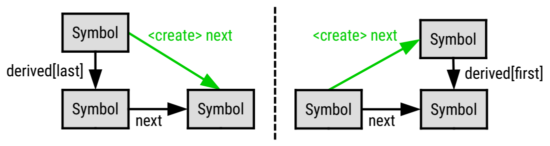

Figure 3. Lifting rules for the bottom-up propagation of `next` relations.

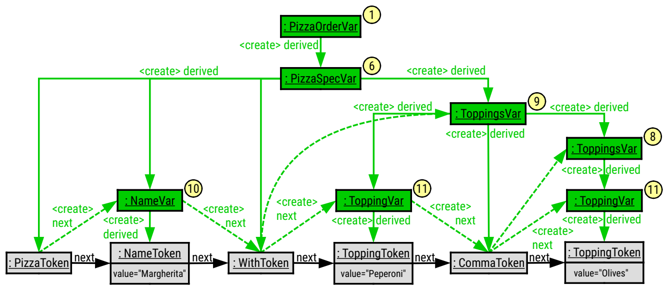

Figure 4. Example parse tree for *"Pizza Margherita with Peperoni, Olives"*. For clarity, the `next` relations are drawn in a dashed style. The numbers in circles refer to the corresponding grammar production rule that created this variable object.

or source models. We conclude this section with a discussion concerning viability and scalability of our envisioned features.

A. Scenario: Rule specification is correct, expected matches in source model do not occur due to defects in the model.

A source model, to which a transformation is to be applied, may contain defects. When this causes the transformation to fail through a lack of matches between the rule specification and the model, developers would like to see the parts of the source model, that are most likely to be those that should be matched, but contain local defects. This allows for a more targeted approach at discovering how the defects have been introduced. As a concrete example to illustrate this scenario, we use the *PizzaSpec* production rule from Fig. 2, applied to a token sequence source model that violates the grammar for the pizza language.

1) Heatmap for Degree of Matching of Source Model Parts: Assuming that matches of structures in the source model fail due to small local defects, developers could be supported by directing their attention to those parts of the model that are closest to a match. To decide which parts to highlight, we define a *degree of matching:* the larger a subgraph of the source model that can be matched to the structure expressed in the rule is, the higher we consider its degree of matching. To visualize this, we highlight the respective parts of the model with color levels akin to a heatmap (see Fig. 5). The "hotter" a structure in the source model, the closer it is to be matched by the rule. This highlights where small edits could be made to the model to enable matches. In this example, the longest and therefore "hottest" partial match is a token sequence with the defect of a missing pizza name. Algorithmically, the partial matches can be computed by considering the rule structure and creating mutants of it, up to a certain edit distance (e.g., add or remove just one object or relation). These mutants are then matched against the source model.

2) Suggested Modifications of Source Model: In addition to knowing where to make edits, it would be helpful for developers to get concrete suggestions of modifications to

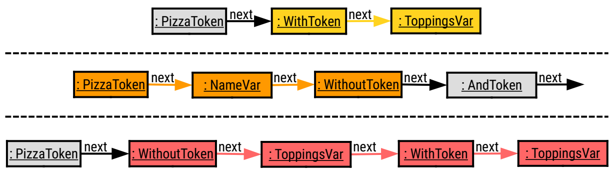

Figure 5. Heatmap for degree of matching of source model parts to the structure expressed in the grammar production rule no. 7 (see Fig. 2).

the source model that would make the transformation rule applicable. Ideally, the recommended modifications should be minimal, local changes to the model. They can be computed by creating mutants of the rule, matching those and then reversing the mutating edits. If the changes concern attribute values, the concrete values or conditions on those values in the rule specification can be considered as a set of constraints to be solved. We propose the visualization shown in Fig. 6. Here, a series of model elements to remove (colored, dashed) or add (colored, solid) are highlighted, with the color depending on a heatmap scheme signifying how extensive the modification would be (smaller edit distance is better/“hotter”). To further restrict the search space, developers could be allowed to express additional knowledge in the local environment of a partial match. They could pin down model elements they are sure must be present, thereby preventing their suggested removal, or conversely forbid the addition of certain elements.

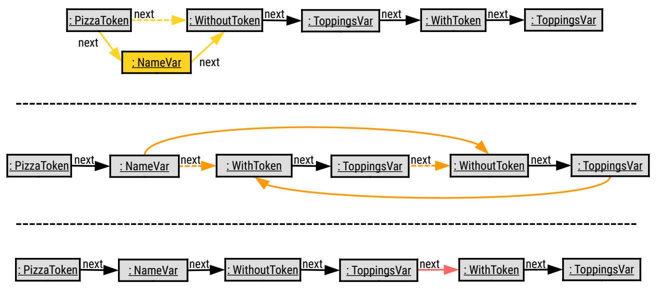

Figure 6. Suggested modifications of the source model with heatmap coloring.

B. Scenario: Source model is correct, expected matches do not occur due to incorrect transformation rule specification.

Conversely, a transformation might not be applicable to a part of the source model because an incorrect rule specification does not match its structure. In this case, developers could be assisted by showing them how to fix the rule, if there is a small edit operation that would enable a match.

1) Suggested Modifications of Transformation Rule: As described in Sec. IV-A1, mutants of the rule specification with small edit distance can be created. These mutants can then be applied to a selected part of the source model. If they match, the edited elements in the mutants can then be assigned a heatmap level proportional to the edit distance. Additions and removals of model elements follow the same graphical syntax as the visualization in IV-A2.

C. Scope and Viability of the Proposed Features

We introduced our proposed features for analyzing and debugging model transformations using a very basic and concise running example with endogeneous transformations. However, experiences from the industrial practice suggest that source models, to which transformations are applied, are often very large, up to millions of nodes. To deal with such extensive models, tools can offer views and filters for displaying only the relevant parts of a model (those with embedded visualizations), supplemented with commands to quickly browse between them (e.g., jump between partial matches, ordered by descending degree of matching). The number of suggested modifications that are generated can be limited by specifying an upper bound on the edit distance to structures that are already present in the artifacts. Also, in general, the source and target models can have different types, so different syntaxes might be involved in rule specifications or source models. If the embedded visualizations are not suited for a concrete graphical syntax, a more appropriate abstract syntax can be used.

V. Conclusion and Future Work

We discussed some common scenarios encountered by model engineers, when they are analyzing or debugging declarative model transformations. To support these recurring tasks, we proposed a series of techniques based on visualizing information embedded into the graphical syntax of transformation rules and source models. We motivated these features by pointing out that they would have the advantage to allow developers to use the same representations for analysis and debugging tasks that they use to create artifacts like models and transformation rule specifications in the first place. This avoids the shortcomings of many established MDE tools, that force the developer to step through the imperative execution of matching algorithms in the transformation engine, and thus impose the mental burden of a paradigm shift.

For future work, we plan to integrate our proposed features into Henshin. Further, we would then like to evaluate them in a user study, to see if there is a quantifiable speedup for solving representative debugging tasks on real world models using these features compared with the existing means for stepwise debugging. Also, we would like to explore how similar visualizations can be used with different concrete syntaxes than the graph-like “box-and-line-languages” we considered so far, e.g., graphical syntaxes that use nesting of elements, or how declarative debugging techniques can be used in combination with imperative ones to draw on the respective advantages of both approaches.

References

[1] M. Biehl, "Literature study on model transformations," *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, vol. 291, 2010.

[2] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A usability-focused framework for EMF model transformation development," in *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*.

[3] M. Lawley and K. Raymond, "Implementing a practical declarative logic-based model transformation engine," in *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*.

[4] M. Hibberd, M. Lawley, and K. Raymond, "Forensic debugging of model transformations," in *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*.

[5] M. Tichy, L. Beaucamp, and S. Kögel, "Towards debugging the matching of henshin model transformations rules," in *Proceedings of MODELS 2017 Satellite Event*.

[6] T. Mészáros, P. Fehér, and L. Lengyel, "Visual debugging support for graph rewriting-based model transformations," in *Proceedings of Eurocon 2013, International Conference on Computer as a Tool, Zagreb, Croatia, July 1-4, 2013*.

[7] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova, "Modelling and analysis using GROOVE," *STTT*, vol. 14.

[8] J. Schönböck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Catch me if you can - debugging support for model transformations," in *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Reports and Revised Selected Papers*.

[9] D. Schilling, "Kompositionale Softwareverifikation mechatronischer Systeme," Ph.D. dissertation, University of Paderborn, Germany, 2006.

[10] A. Rensink, "Explicit state model checking for graph grammars," in *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*.