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

Florian Ege, Matthias Tichy

Institute of Software Engineering and Programming Languages
Ulm University

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 incorrect models or transformations (cf. [3]). 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

Motivation

"Conceptual break" in the context of debugging declarative model transformations with graphical syntax:

level of abstraction and semantics that developers work on
vs.
level that is used for debugging

Related Work

- Stepwise debugging, analogue to debuggers for imperative languages
  - breakpoints
  - visualization of intermediate states of matching
  - rollbacks
  - transformation of transformations to other formalisms (e.g., transformation nets)

  … but all these expose engine internals:
  imperative graph matching algorithm, sequence of low-level operations
- Model checking, verification of properties

  … rather analytical than constructive (in terms of actually fixing defects)

Idea: Embedded Visualizations

Facilitate analysis and debugging by presenting information integrated in the graphical syntax of instance models and transformation rule specifications

→ avoid conceptual break by staying on the same level of abstraction
→ develop and debug using the same graphical syntax

Running Example: Pizza DSL

⟨*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)

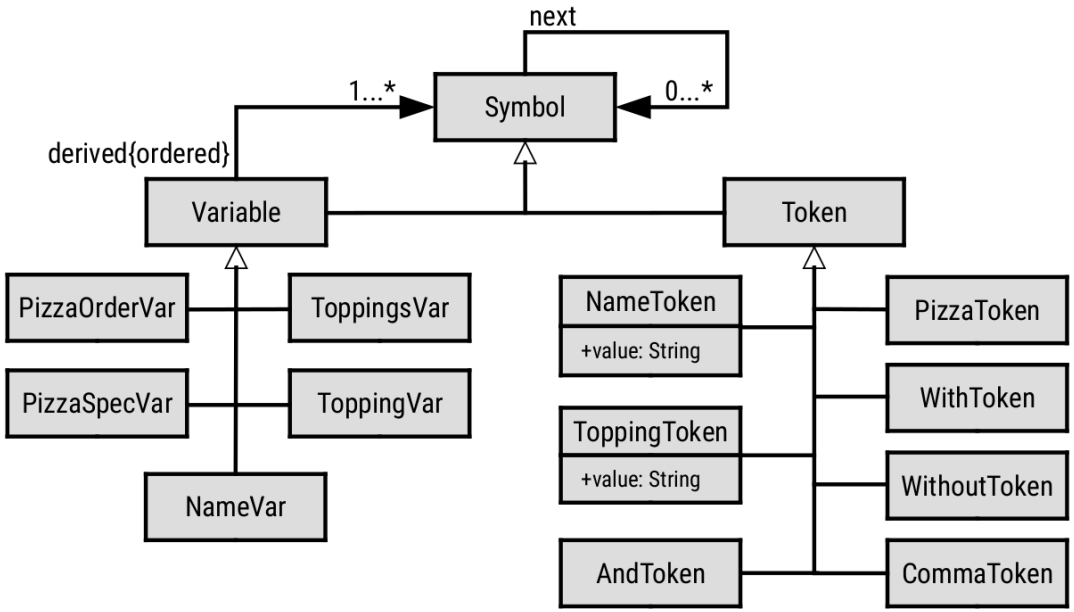
⟨*Toppings*⟩ ::= ⟨*Topping*⟩ (8)
| ⟨*Topping*⟩ **,** ⟨*Toppings*⟩ (9)

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

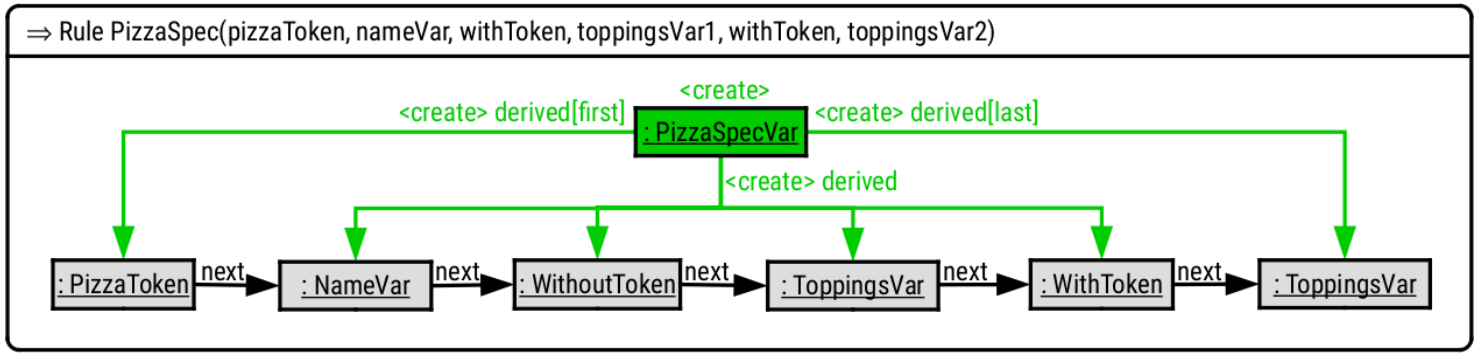
⟨*Topping*⟩ ::= **Artichokes** | **Garlic** | **Olives** | **Peperoni** | **...** (11)

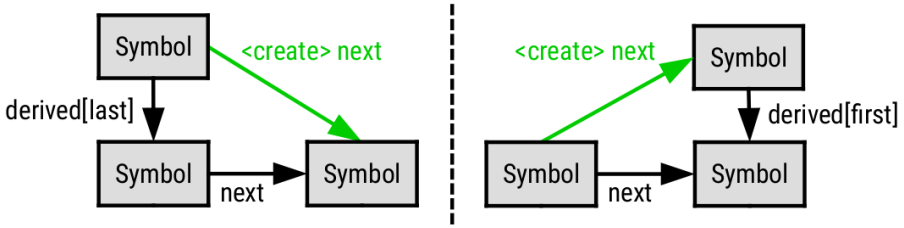
This grammar derives sentences like the following:

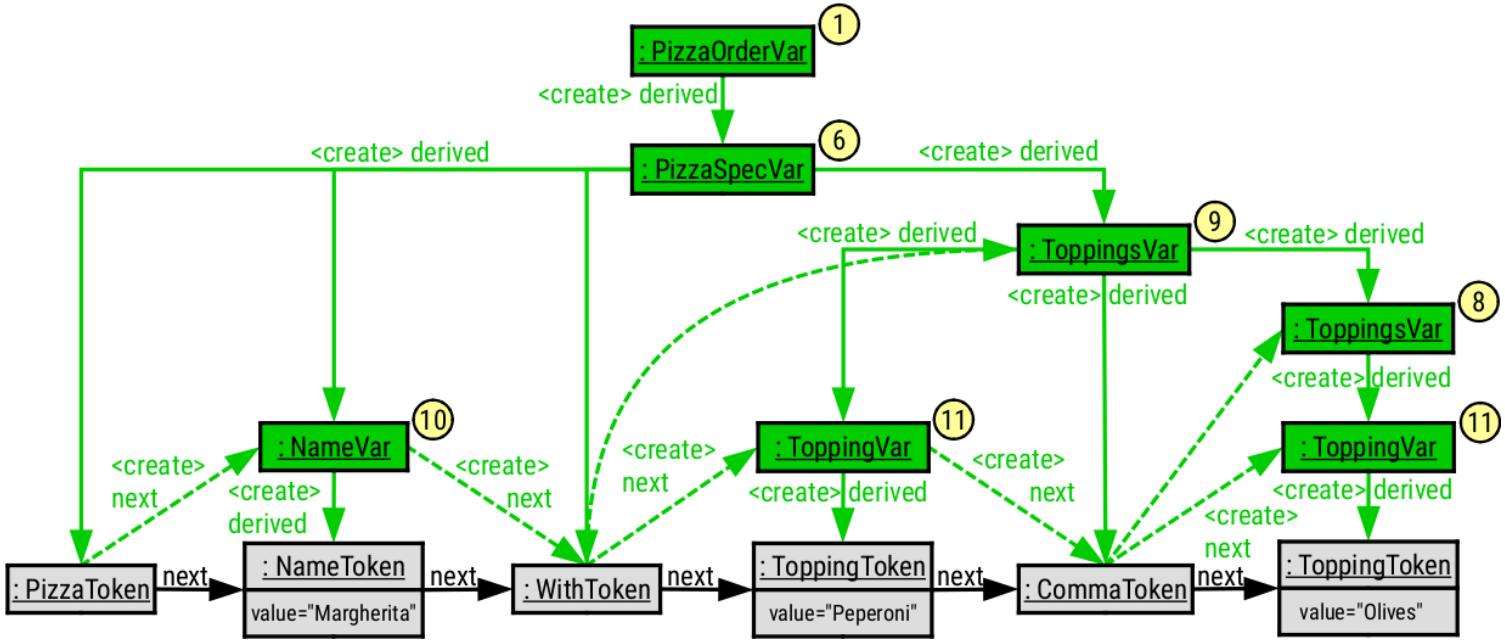
- *Pizza with Mushrooms, Onions, Parmigiano*
- *Pizza Contadina without Asparagus*
- *Pizza Funghi without Parsley, Olives with Ruccola and Pizza Hawaii without Ananas with Mozzarella, Oregano*



Running Example: Pizza DSL





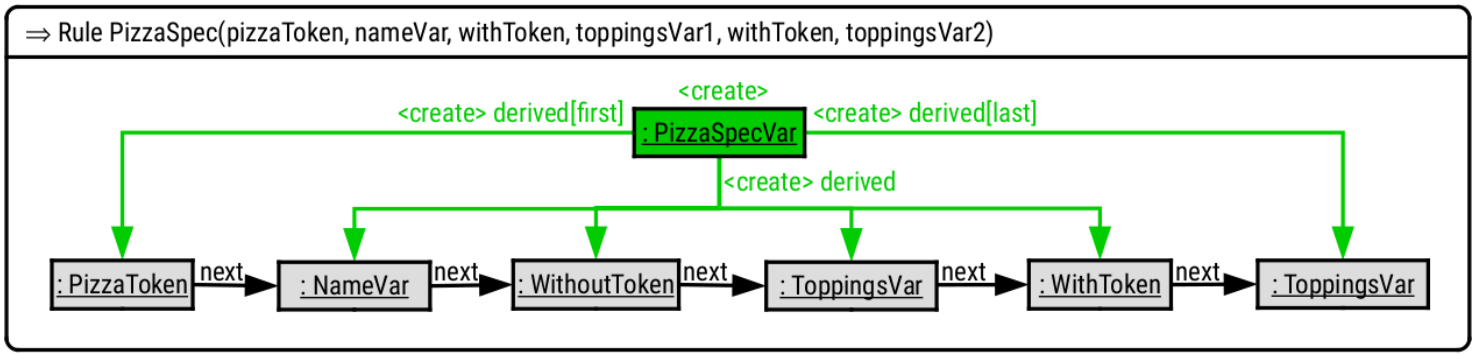


Debugging Scenarios

- Rule specification is correct, expected matches in source model do not occur due to defects in the model
- Instance model is correct, expected matches do not occur due to incorrect transformation rule specification

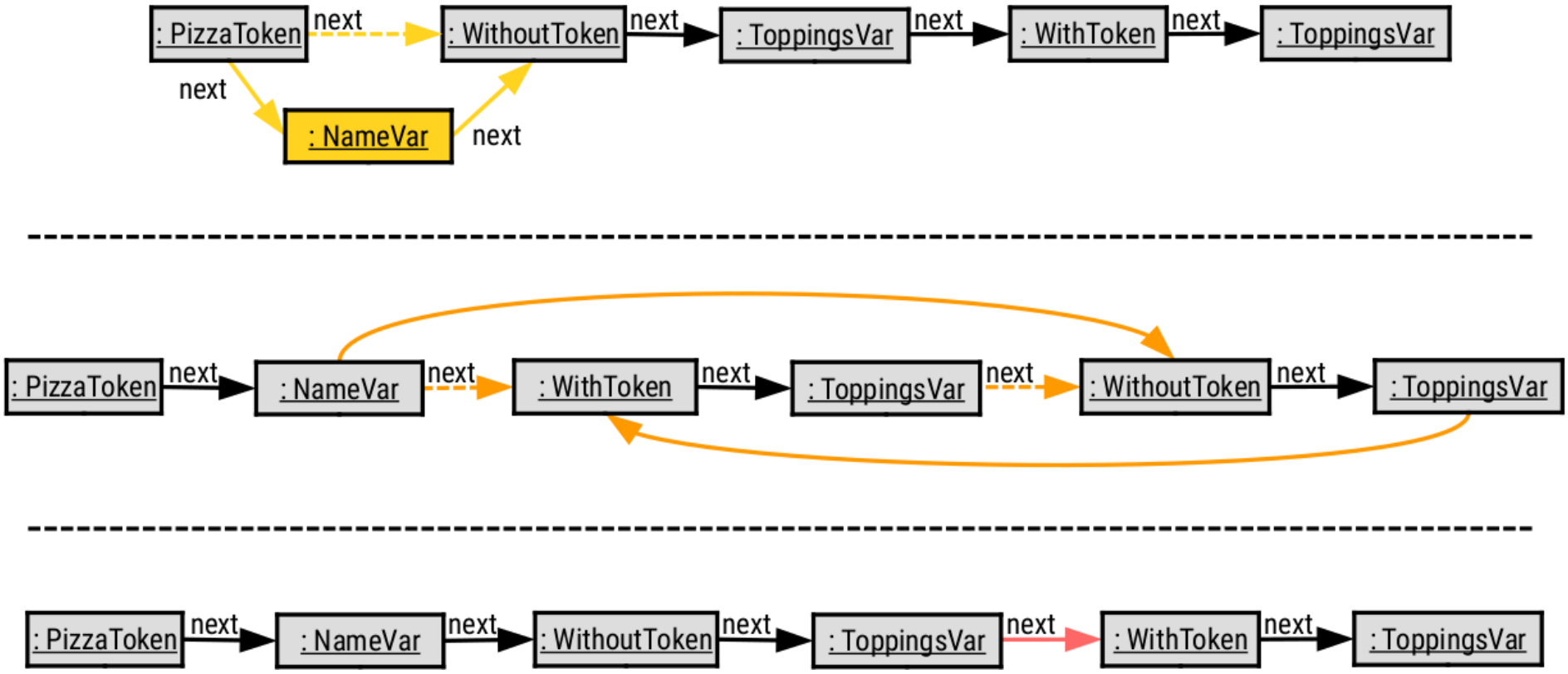
Incorrect Instance Model

- Heatmap for Degree of Matching
  - suggests parts that "almost match" → potential location for fix



Incorrect Instance Model

- Suggestion of modifications of the instance model (with heatmap)
  - find "minimal" fix that makes rules applicable



Incorrect Rule Specification

- … well, basically analogue approaches
  - highlighting parts of rule that would match (i.e. be applicable)
  - suggestions of edits for left-hand side of rule to make it match
  - heatmap coloring for degrees of matching or closeness of edit distance

Validity Considerations

- Running example is very basic
  - cognitive scalability?

- Real-world models are often quite large
  - features for views/filtering to reduce number of displayed elements
  - support for browsing between relevant parts of model

Future Work

- Integration of proposed features into Henshin
  - User Study:
    Speedup for representative debugging tasks vs. existing approaches?

- Considered so far: "Box-and-Line" languages
  - transfer to different graphical syntaxes, e.g., with nested structures

- Other visual cues
  - indicators for numbers of matches
  - repeatability of transformations
  - …