Two MontiCore solutions to the TTC 2025 UVL to Dot case

Alex Lüpges[1], Janik Rapp[1] and Bernhard Rumpe[1]

[1]*Software Engineering, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany*

Abstract

This paper presents two solutions to the TTC 2025 "UVL to Dot" case using the MontiCore language workbench, applying either a handwritten string construction using the visitor design pattern or pattern matching using the concrete syntax within templates.

Keywords

Model-Driven, Modular Model Transformations, Domain-Specific Transformation Languages

1. Introduction

As generative AI continues to advance rapidly, it is increasingly being considered for supporting—or even fully automating—a wide range of tasks, including model transformations. However, due to the specialized nature of dedicated model transformation languages, there are few examples that large language models can use to learn. As a result, large language models tend to perform better when generating transformation code in general-purpose programming languages than for specialized model transformation languages.

The *Universal Variability to Dot* [1] case of the Transformation Tool Contest (TTC) 2025 aims to make a contribution to this by explicitly implementing the transformation from the textual Universal Variability Language (UVL, [2]) to the textual domain-specific language (DSL) DOT. The focus on textual modeling languages should therefore enable a comparison between well-known transformation tools and large language models (LLMs).

In this paper, we present two solutions using the language workbench MontiCore [3] and its extension MontiTrans [4]. MontiCore enables the definition of the UVL grammar for parsing existing models. The first solution then uses visitors to build a textual DOT representation, while the second solution uses MontiTrans' pattern matching to create DOT models from FreeMarker templates.

2. MontiCore

MontiCore [3] is a language workbench for the development and composition of textual DSLs. Starting with a context-free grammar, MontiCore generates a variety of infrastructure to parse, represent, and process models. This includes a parser, symbol tables, and visitors, as well as frameworks for model checking and code generation. The parsing process is done in two stages. At first, the underlying ANTLR parser constructs the parse tree, that is transformed to a language-specific internal representation of the Abstract Syntax Tree (AST) afterwards. This AST can then be navigated using MontiCore's visitor infrastructure. Although MontiCore uses various text-to-model, model-to-model, and model-to-text transformations, all of these are implemented exclusively through specific Java code. Language-specific support for external graph or model transformations is not provided.

MontiTrans [4] extends MontiCore with support for the development of domain-specific transformations via derived domain-specific transformation languages (DSTLs). DSTLs enable the definition of transformations within modeling languages by extending their concrete syntax with transformation
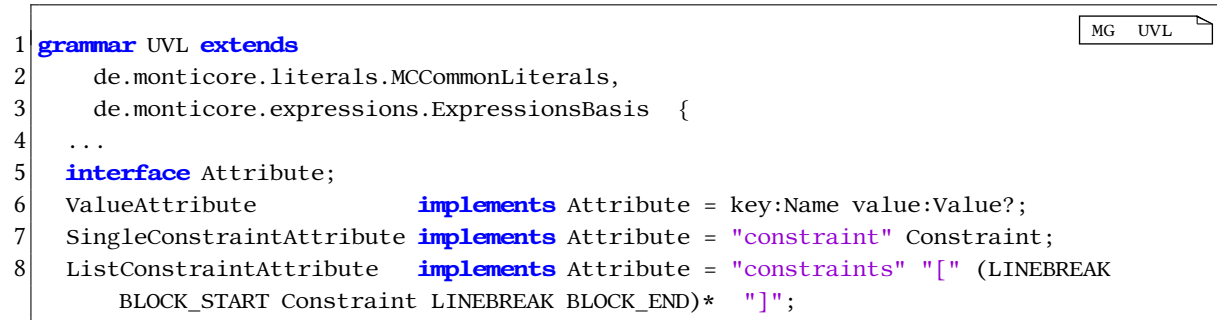
directives. This enables modelers and domain experts to describe transformations using the syntax of their familiar modeling language without having to learn an additional general transformation language. However, while general transformation languages only have to be developed once, DSTLs have to be developed separately for each modeling language. MontiTrans can therefore automatically derive and generate DSTLs from given modeling languages. MontiTrans uses a model sensitive, search plan-driven local search for pattern matching when executing a transformation, regardless of whether the transformation is interpreted or executed as generated code. The search plan is therefore influenced both by the information of a pattern and by the information on the model to be searched. Finally, backtracking is used to reduce the candidate sets as much as possible.

3. Solution Overview

3.1. Domain-Specific Language

The challenge provides ANTLR and TreeSitter grammars, as well as models conforming to an Ecore metamodel. Since MontiCore uses its own grammar format, we must first create a MontiCore grammar for the Universal Variability Language.

Our UVL grammar uses two component grammars from the MontiCore library: MCCommonLiterals provides literals and ExpressionsBasis provides the expressions used within the constraints. Further component grammars are used transitively.

```
                                                              MG   UVL
1  grammar UVL extends
2      de.monticore.literals.MCCommonLiterals,
3      de.monticore.expressions.ExpressionsBasis  {
4  ...
5  interface Attribute;
6  ValueAttribute              implements Attribute = key:Name value:Value?;
7  SingleConstraintAttribute implements Attribute = "constraint" Constraint;
8  ListConstraintAttribute   implements Attribute = "constraints" "[" (LINEBREAK
       BLOCK_START Constraint LINEBREAK BLOCK_END)*  "]";
```

We demonstrate language composition using UVL attributes as an example. UVL attributes are defined using a nonterminal interface. Then, we define three implementations of an attribute: ValueAttribute, SingleConstraintAttribute, and ListConstraintAttribute, all implementing the Attribute nonterminal interface. In terms of concrete syntax, these three implementations are all valid alternatives for an Attribute. Although this case's UVL language is not intended for reuse in other languages, interface productions enable the black-box reuse or refinement of a language's elements. We opted not to define the symbols and scopes of the language, which would turn the AST into a graph. MontiCore's language library is tailored for Java-like languages, resulting in the need for a preprocessing step converting changes in the indentation level into BLOCK_START and BLOCK_END tokens.

3.2. Solution 1: Visitor

The first solution is similar to the example solution in that it traverses the AST and writes the DOT graph into a string builder using plain old Java. It differs from the example solution by using the MontiCore AST constructed by our DSL's parser. We have opted to include this solution as it provides a baseline for the second, pattern matching, solution. Similarly, it demonstrates the usage of expressions in constraints from MontiCore's language component library. Since Java does not offer built-in support for runtime dispatching, MontiCore's visitor infrastructure performs a double dispatch to support language composition. Because we designed the language so that attributes can easily be extended via language composition, this double dispatch is necessary.

To output the edges of feature groups, we traverse the AST with a custom traversal strategy for the ASTGroups, of which an excerpt is shown below:

```java
                                                                              Java
1  @Override
2  public void traverse(ASTGroup node) {
3   for (var subFeature : node.getGroupSpec().getFeatureList()) {
4    featureStack.push(subFeature);
5    subFeature.accept(getTraverser()); // continue to traverse the sub feature
6    featureStack.pop();
7    sb.append(featureStack.peek().getRef().asString());    // access parent reference
8    sb.append(" -> ");
9    sb.append(subFeature.getRef().asString());              // access subFeature reference
10   if (node.isAlt())
11    sb.append("[arrowhead=\"none\", arrowtail=\"odot\", dir=\"both\"] \n");
12   else if (node.isOr())
13    sb.append("[arrowhead=\"none\", arrowtail=\"dot\", dir=\"both\"] \n");
14   else if ...
15   }
16  }
17 });
```

For each group we visit, we push each sub-feature to a stack and traverse the sub-feature's children depth-first. Next, we create an edge from the parent feature accessed via the stack to the current sub-feature. The edge's style is selected based on the group's type.

3.3. Domain-Specific Transformation Language

Using a DSTL derived by MontiTrans, model developers are able to describe patterns and integrated replacements in the already known DSL's syntax, with minimal additional syntax elements.

Given a production of the UVL grammar like `Namespace`,

```
                                                                          MG   UVL
1   Namespace = "namespace" Reference;
```

a production `Namespace_Pat` is derived which is used to describe the pattern of a namespace.

```
                                                                        MG   UVLTR
1   Namespace_Pat implements ITFNamespace =
2   "namespace" reference:ITFReference ||  // concrete syntax
3   "Namespace" schemaVar:Name        ||  // capture + abstract syntax
4                                         // capture + concrete syntax
5   (("Namespace"? schemaVar:Name) | ("Namespace" schemaVar:Name?))
6     "[[" "namespace" reference:ITFReference "]]";
```

The first option is to match against the concrete syntax of a model element. For example, `namespace` . . . with . . . being a pattern for a reference. Any names or identifiers within this concrete syntax can be generalized using schema variables, which are prefixed with $. The second alternative allows for matching based on the abstract syntax of a model element, i.e., its type. In this example, `Namespace` $N is a valid capturing pattern. Additionally, the matched model element is captured and made accessible via a schema variable. The third alternative allows the capture of a model element based on a concrete syntax pattern. $N [[namespace . . .]] is a valid example of this alternative.

Further implementations of the `ITFNamespace` interface production consist of an optional pattern, a negated pattern, a list of inner patterns, and a replacement operator.

```
                                                                        MG   UVLTR
1   Namespace_Opt implements ITFNamespace =
2    "opt" "[[" namespace:ITFNamespace "]]";
```

All implementations of an interface are valid alternatives when it comes to the concrete syntax. For this case, only the pattern and optional pattern derivations are used. By its nature, a DSTL only supports homogeneous transformations of models of its DSL. To achieve heterogeneous model-to-model

transformations, the original and target DSLs must be composed and the transformations be described using the composed DSTL. Instead, our solution relies on MontiCore's templating engine, which uses FreeMarker, to achieve a UVL-model-to-DOT-text transformation.

3.3.1. Template 1: Creating the DOT Graph

The first template and our de facto entry point prints the properties of the graph and includes the next three templates.

```
                                                              FTL   01CreateDot
1 digraph FeatureModel {
2   rankdir="TB"
3   ...
4   ${tc.include("templates.02CreateNodes")};
5   ${tc.include("templates.03Groups")};
6   ${tc.include("templates.04Constraints")};
7 }
```

MontiCore's templating engine supports tracing of model elements and the templates used for generation. These features serves as building blocks for supporting incremental builds.

3.3.2. Template 2: Creating Nodes from Features

The second template creates a box node for each feature. Abstract features result in an invhouse shape. Although the special shape for abstract features was not described in the case description, we have opted to follow the reference implementation in this regard. This also showcases the optional directive embedded within the DSTL.

To develop a pattern, one usually starts from a concrete example model [4]. In our case, we want to match a feature that may or may not have the *abstract* value-attribute. An example of such a feature can be seen below:
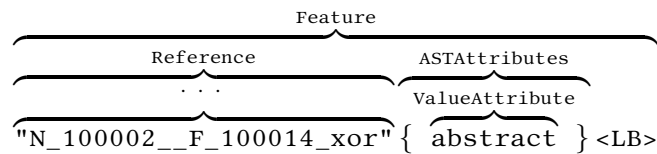
To increase readability, a partial parse tree is added to this example. Next, the pattern is generalized by replacing concrete names with schema variables and, in our case, marking the attributes as optional. This generalized pattern can then be used within a template:

```
                                                              FTL   02CreateNotes
1 <#list pm.match("Reference $FName opt[[$A[[{abstract}]] ]]<LB>",ast) as match>
2   ${match['$FName'].get().asString()}
3   [fillcolor="#ABACEA" tooltip="Cardinality: None" shape="${match['$A'].isPresent()?
        then("invhouse","box")}"]
4 </#list>
```
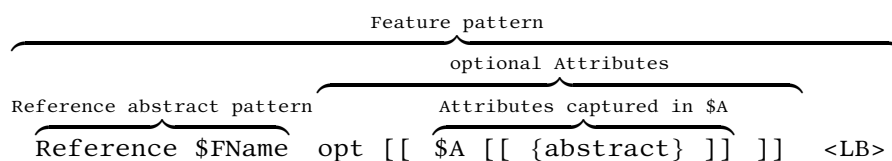
The template uses the pm.match(String pattern) utility method to define and use a pattern during template generation.

The pattern matches a feature based on the feature's concrete syntax. First, the feature's reference is captured in the schema variable $FName. This schema variable is later used to print the feature's

name again. While we could capture the individual parts of the reference, we chose this alternative for performance reasons. Next, an optional value attribute with the key `abstract` is matched and captured in the $A schema variable. If this schema variable contains a matched model element - that is, if the feature was marked as abstract - we select another shape for the node. Behind the scenes, the pattern, which is a textual model of the DSTL, is parsed and converted to an object diagram, as can be seen in Figure 1.
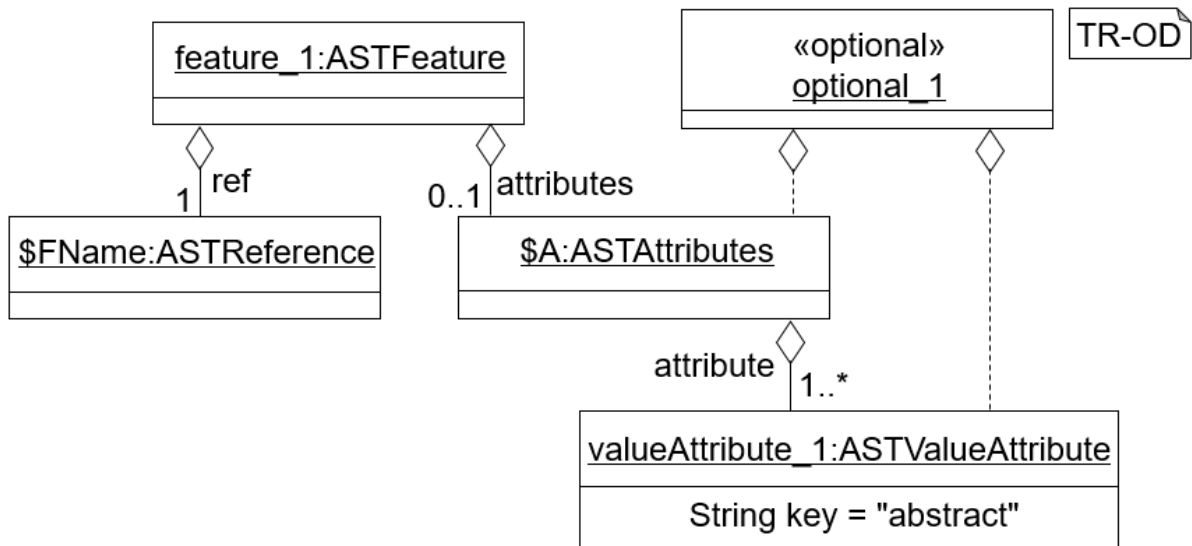

Figure 1: The pattern of the node creating template in object diagram notation.

From this pattern, a search plan is calculated: Starting with finding a possible candidate for `feature_1`, i.e., all existing features, the candidate's `ref` is used as a candidate for $FName. Next, the engine attempts an optional match: A candidate for the `ASTAttributes` object $A is selected from the feature candidate's attributes and an `ASTValueAttribute` with `abstract` as its key matched. If no candidates can be found for the attribute, the engine performs backtracking until no more candidates can be found or until the backtracking reaches the optional step. In case no model element matches the optional pattern, the engine continues without a match for the schema variable $A.

3.3.3. Template 3: Creating Edges from Groups

The third template translates feature groups into edges by matching an outer feature, its Group, as well as an inner feature. We capture the outer feature's reference in `FRef`. Next, we match and capture any kind of group. The tokens `<LB>`, `<BS>`, and `<BE>` are used to describe line breaks and indentation changes in the model.

```
                                                      FTL   03Groups
1  <#list pm.match("
2  Reference $FRef <LB> <BS>
3   Group $G [[  $_ <LB> <BS>
4     Feature [[  Reference $IRef <LB> ]] <BE>
5          ]]  <BE> ", ast) as match>
6     ${match['$FRef'].get().asString()}
7     ->
8     ${match['$IRef'].get().asString()}
9     <#if match['$G'].get().isAlt()>
10       [arrowhead="none", arrowtail="odot", dir="forward"]
11    ...
```

Due to the limited set of feature group kinds, we have opted to use a constant group in our UVL grammar. Currently, we thus have to extend the group pattern of the UVLTR grammar by allowing the `$_` placeholder in place of the constants, such as `alternative`, `or`, etc. This avoids the need to repeat the pattern search for each kind of group. Using an interface instead of a constant group might have been the better language design choice, but it would have increased the size of the search plan. The DSTL derivation process of MontiCore does not support this by default. However, the extensibility of the derivation process allows us to include hand-written additions to the DSTL.

The final template iterates over the constraints and adds the result of a model-to-text transformation to the table. This transformation uses an automatically generated *pretty printer*, reverting the parsing process.

4. Evaluation

The first solution is comparable to the reference solution, just using language composition for rapid language engineering and a different programming language. Although a large language model could generate the model-to-text transformation class, developing it requires knowledge of UVL's abstract syntax. The compositional nature of the DSL incurs a performance penalty in comparison [5].
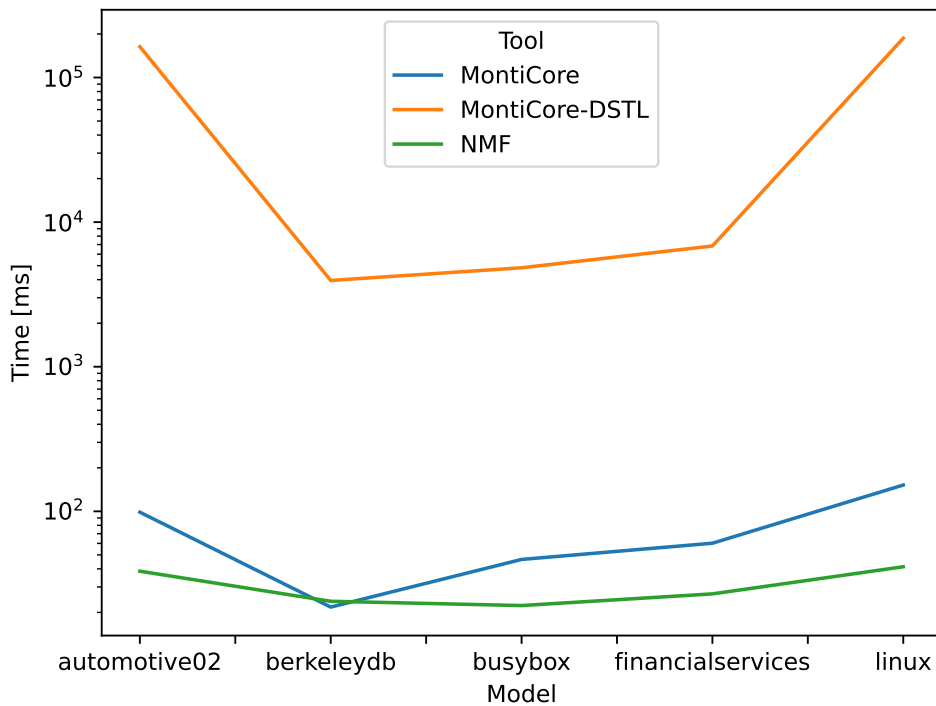

Figure 2: Benchmark of the initial phase, in which the initial UVL model is processed. The time axis is shown on a log scale.

One of the strengths of pattern matching using the concrete syntax is that domain experts can describe patterns and transformations using established vocabulary of their domain and without the accidental complexity of general transformation languages [6]. However, in the case of UVL, a language with few keywords and indentation sensitivity, the patterns are difficult to understand. Other languages, such as a textual notation of UML class diagrams [4], UML statecharts, and MontiArc automata [7], have shown better results.

A better designed MontiCore UVL grammar or a different way to represent indentation sensitivity within the UVL-DSTL may contribute to an improved experience. Similarly, supporting transformation

developers with a designated editor should improve the experience. One could experiment if large language models using a structured output in a DSTL strikes a good middle ground between understandable model transformations for domain experts and AI-assisted transformation development. We initially planned to evaluate LLMs in their ability to write transformation using DSTLs and few-shot learning. However the surprising complexity of the patterns due to UVL's context sensitivity made this infeasible.

MontiTrans only supports pattern matching and homogeneous model-to-model transformations, i.e. within the same DSL. To achieve heterogeneous transformations (e.g., from UVL to DOT) one could compose the source and target languages and write transformations within this new composed language. However, we have opted to use the FreeMarker template engine to create model-to-text transformations by pattern matching the UVL model and writing DOT models.
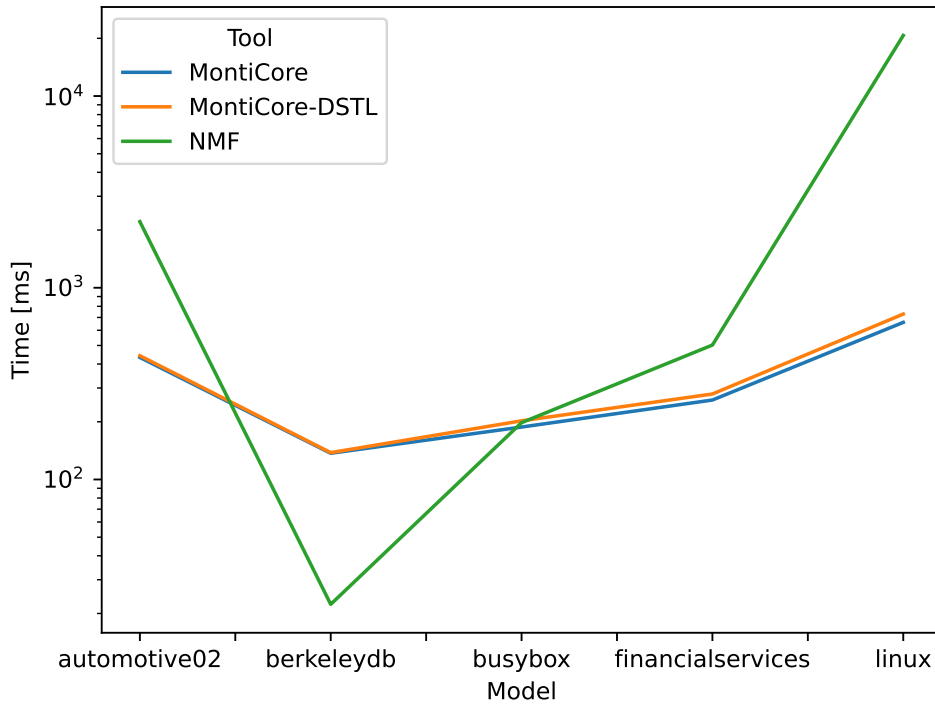

Figure 3: Benchmark of the load phase, in which the models are parsed. The time axis is shown on a log scale.

In regards to performance, the interpretative nature of our second solution is evident in Figure 2, with the *MontiCore-DSTL* tool of the second solution taking a much longer time. Although MontiTrans provides support for pre-compiled, i.e. generated, transformations, we determined that the concept of interpreted patterns defined within FreeMarker templates was better suited to this challenge. The pattern matching part of MontiCore [4] has been laying dormant for some time and still offers areas for performance improvements. This, in combination with the interpreter overhead, results in non-optimal performance results. Recently, however, we have improved our generated parsers to be able to efficiently parse transformation rules. Figure 3 shows the time needed for the load phase of both MontiCore solutions and the NMF reference solution. The slightly longer duration of the DSTL solution is caused by the initialization of the interpreter. While the transformation engine reports on the model elements that were matched and modified, incremental transformations are not yet supported.

5. Conclusion

This paper has presented two solutions to the "UVL to Dot Case" [1], both using the MontiCore language workbench. The first solution is a standard visitor-based string printer and the second solution provides an example of using domain-specific pattern matching in a templating language without inbuilt pattern matching support. The case revealed some of the shortcomings of indentation-aware DSTLs and identified areas for future development.

Acknowledgments

Declaration on Generative AI

During the preparation of this work, the authors used DeepL in order to: Grammar and spelling check, Paraphrase and reword. After using this service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] G. Hinkel, S. Greiner, T. le Calvar, Universal Variability to Dot (2025). URL: https://github.com/TransformationToolContest/ttc2025-live/blob/main/docs/UVL_to_Dot_TTC_2025_Live_Contest.pdf.

[2] D. Benavides, C. Sundermann, K. Feichtinger, J. A. Galindo, R. Rabiser, T. Thüm, UVL: feature modelling with the universal variability language, Journal of Systems and Software 225 (2025) 112326. doi:10.1016/J.JSS.2024.112326.

[3] K. Hölldobler, B. Rumpe, MontiCore 5 Language Workbench Edition 2017, Aachener Informatik-Berichte, Software Engineering, Band 32, Shaker Verlag, 2017. URL: http://www.se-rwth.de/publications/MontiCore-5-Language-Workbench-Edition-2017.pdf.

[4] K. Hölldobler, MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen, Aachener Informatik-Berichte, Software Engineering, Band 36, Shaker Verlag, 2018. doi:10.18154/RWTH-2019-00468.

[5] N. Jansen, A. Lüpges, B. Rumpe, Lessons Learned from Developing the MontiCore Language Workbench: Challenges of Modular Language Design, in: Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering, 2025, pp. 112–127. doi:10.1145/3732771.3742717.

[6] R. France, B. Rumpe, Model-driven Development of Complex Software: A Research Roadmap, Future of Software Engineering (FOSE '07) (2007) 37–54. URL: http://www.se-rwth.de/publications/Model-driven-Development-of-Complex-Software-A-Research-Roadmap.pdf.

[7] K. Adam, K. Hölldobler, B. Rumpe, A. Wortmann, Modeling Robotics Software Architectures with Modular Model Transformations, Journal of Software Engineering for Robotics (JOSER) 8 (2017) 3–16. doi:10.1109/IRC.2017.16.