Universal Variability to Dot

The TTC 2025 Live Contest

Georg Hinkel
georg.hinkel@hs-rm.de
RheinMain University of Applied Sciences
Wiesbaden, Germany

Sandra Greiner
greiner@imada.sdu.dk
University of Southern Denmark
Copenhagen, Denmark

Theo le Calvar
theo.le-calvar@imt-atlantique.fr
IMT Atlantique
Nantes, France

ABSTRACT

In times of large language models (LLMs) being able to generate code easily on the request of a user, one may doubt the usefulness of model transformation languages. Therefore, we propose a case study to compare LLM-generated model transformations with manually created model transformations. For this, we chose a case study that makes it easy for an LLM to generate a solution and created a benchmark environment to evaluate different solutions with regard to performance.

KEYWORDS

incremental, model-driven, transformation

1 INTRODUCTION

Many model transformation languages have not many users and there is typically not a large amount of model transformation code that is available as training data for an LLM. A possible consequence of this could be that LLMs are less able to generate model transformation code than they are able to generate code in general-purpose programming languages. This could be another argument against the usage of dedicated model transformation languages in the first place.

With this case study held at the Transformation Tool Contest (TTC), we want to analyze how far LLMs are already in generating code for model transformation problems. Because the same arguments against model transformation could also apply for model-driven engineering in general, we chose a model transformation problem that starts and ends with a textual DSL, such that the LLM does not have to analyze the XMI representation of a model. Our goal is to use this benchmark to evaluate how good AI tools are already in solving model transformation problems.

The remainder of this paper is structured as follows: Section 2 discusses the challenges focused by this case study. Section 3 describes the model transformation scenario. Section 4 introduces the benchmark framework. Section 5 briefly introduces the reference solution. Section 6 discusses related work.

2 CHALLENGES

We believe that the case study should not be that challenging for many model transformation tools, which is why we believe that the task could be doable for LLMs. The interesting bits of the benchmark thus concern solutions partially or entirely generated by AI.

In particular, the following aspects of the problem are especially important:

- How good are transformations generated by an LLM?
- How many changes are necessary to adjust transformations generated by an LLM?
- What prompts can be used to generate a transformation?
- Can LLMs generate code to support features exposed by model transformations, such as in particular incremental change propagation?

We ask all authors of solutions that took advantage of generative AI to comment on how their tool copes with the challenges described above.

For solutions that are written entirely by humans, we ask solution authors to describe the advantages their tool brings compared to the reference solution. These advantages can be in any direction such as, e.g., maintainability, conciseness, or incremental change propagation.

Besides the transformation, we also welcome solutions that use different parsing technologies or use AI-generated code to parse the source UVL files or propagate the diffs incrementally.

3 CASE DESCRIPTION

In this section, we introduce the source language UVL in Section 3.1, the target language Dot in Section 3.2, sketch the transformation in Section 3.3 and discuss incremental change propagation in Section 3.4.

3.1 The Universal Variability Language

The Universal Variability Language (UVL, [2]) is a standardized language to specify feature models. It is designed to be human-readable.

```
features
    A
        mandatory
            B {abstract}
                or
                    C
                    D
            E
                alternative
                    F
                    G
constraints
    A & B
    C | (!B | D)
    E => G
    A <=> B
```

Listing 1: Example of a UVL feature model

An example of a UVL file is shown in Listing 1. It consists of a feature A that has two mandatory child features B and E where B can be either C or D and E has the alternatives F and G. Furthermore, there are a few constraints, depicted as boolean formulas.

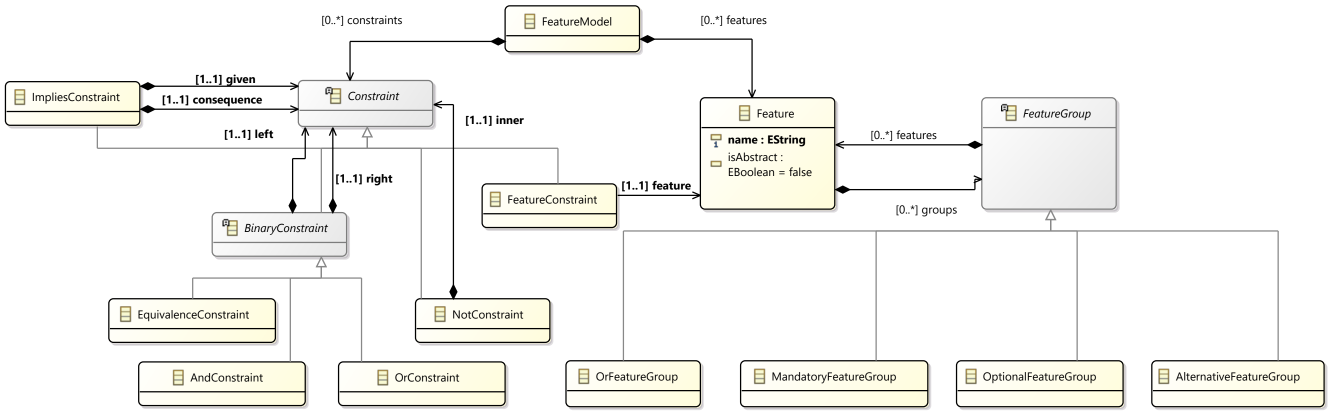

Figure 1: UVL metamodel used in the benchmark framework

UVL also supports annotating features where annotations can have numeric numbers that can be constrained, but these features are not used in the scope of the benchmark. Rather, the benchmark uses a range of industry-scale models provided by [9]. The language features used are all the syntax features from Listing 1, plus *optional* feature groups.

From a language engineering point of view, the grammar of UVL is an indentation-aware language, since the language uses indentation to decide to which feature group a given feature belongs to. In the example of Listing 1, the indentation is used to find that E in line 8 belongs to the mandatory feature group from line 3 rather than the or feature group from line 5. The constraints model semantically requires left recursion (or a refactoring thereof). There are a couple of libraries in different programming languages that can read UVL files. Alternatively, grammars in ANTLR and TreeSitter exist and are available as open source. For the benchmark, we also created a simplified AnyText grammar[1].

The metamodel used in the scope of the benchmark is shown in Figure 1. The models are available in UVL and in XMI according to the metamodel in Figure 1. The metamodel is available in Ecore and NMeta format.

3.2 Dot

Dot is a language defined by Graphviz[2]. It is widely used to describe Graphviz nodes in order to render diagrams. In the scope of this benchmark, the goal is to reproduce the rendering of UVL feature models as defined in the UVL extension. Here, the advantage is that the layout calculation is left to Graphviz, which makes the transformation much easier as it does not have to bother with layout calculations.

The grammar of Dot is shown in Table 1. A dot file can represent either a graph or a directed graph (using the keyword `digraph` for the latter). It contains attribute assignments, node definitions, edge definitions, or subgraphs. Attribute assignments can relate to attributes of nodes, edges, graphs or the entire file and set defaults when nothing else is specified.

[1]The simplification does not support features not exposed by the example models. In addition, we did not implement operator precedence rules in the constraints
[2]https://graphviz.org/doc/info/lang.html

As an alternative to the textual concrete syntax, we also provide an Ecore metamodel for Dot.

3.3 Transformation Rules

The transformation from the UVL to Dot is inspired by the transformation from UVL to Dot that is included in the Visual Studio Code extension for UVL. In the scope of the contest, we set the following properties for the entire directed graph:

- The *rankdir* is set to *TB*.
- The *newrank* is set to *true*.
- The *bgcolor* is set to *#1e1e1e*.
- The *color* of edges is set to *white*.
- The *style* of nodes is set to *filled*, the *fontcolor* is *white* and the *fontname* is *Arial Unicode MS, Arial*.

The name of the directed graph is the name of the UVL file.

Then, for each feature of the feature model, the graph contains a node statement defining a node for this feature. The nodes have the following properties:

- The *fillcolor* is *#ABACEA*.
- The *shape* is *box*.
- The *tooltip* reflects the cardinality of the feature, by default it is *Cardinality: None*.

Feature groups translate to edges as follows:

- If a feature is contained in an alternative group of its parent feature, the Dot file shall have an edge from the parent feature to the child feature with an empty arrow head (*none*) and a hollow arrow tail (*odot*).
- If a feature is contained in an or group of its parent feature, the Dot file shall have an edge from the parent feature to the child feature with an empty arrow head (*none*) and a filled arrow tail (*dot*).
- If a feature is contained in a mandatory group of its parent feature, the Dot file shall have an edge from the parent feature to the child feature with a filled arrow head (*dot*) and an empty arrow tail (*none*).
- If a feature is contained in an optional group of its parent feature, the Dot file shall have an edge from the parent

graph	:	[**strict**] (**graph** | **digraph**) [ID] **'{'** stmt_list **'}'**
stmt_list	:	[stmt [';'] stmt_list]
stmt	:	node_stmt
	|	edge_stmt
	|	attr_stmt
	|	ID '=' ID
	|	subgraph
attr_stmt	:	(**graph** | **node** | **edge**) attr_list
attr_list	:	**'['** [a_list] **']'** [attr_list]
a_list	:	ID '=' ID [(';' | ',')] [a_list]
edge_stmt	:	(node_id | subgraph) edgeRHS [attr_list]
edgeRHS	:	edgeop (node_id | subgraph) [edgeRHS]
node_stmt	:	node_id [attr_list]
node_id	:	ID [port]
port	:	**':'** ID [**':'** compass_pt]
	|	**':'** compass_pt|
subgraph	:	[**subgraph** [ID]] **'{'** stmt_list **'}'**
compass_pt	:	**n** | **ne** | **e** | **se** | **s** | **sw** | **w** | **nw** | **c** | _

Table 1: Grammar of Dot. Terminals are in bold.

feature to the child feature with a hollow arrow head (*odot*) and an empty arrow tail (*none*).

The constraints should be rendered as an HTML table where each constraint is a separate row with a single column. Although the Dot table contains the constraints in exactly the same syntax as UVL, transformations should not simply copy the constraints but render them from the model.

3.4 Change Propagation Rules

We included the changes between the UVL files in Git patch format in the repository. Solutions can choose whether to load the changed feature model (in UVL or XMI format) or to propagate the changes.

4 BENCHMARK

We provide a benchmark framework that can automatically compile, run and check solutions and generate diagrams to analyze the results. The benchmark framework is based on the Train Benchmark [10] and the adjustments made for the case of TTC 2021 incremental laboratory workflows benchmark [6]. The framework as well as metamodels, input models and the reference solution is publicly available online at https://github.com/TransformationToolContest/ttc2025-live.

In the remainder of this section, we first describe the phases of the benchmark in Section 4.1, then explain how to run it in Section 4.2. Next, we introduce the input models in Section 4.3. Section 4.4 explains the correctness checks and Section 4.5 introduces the evaluation criteria for solutions. Section 4.6 then explains the procedure to add a solution to the benchmark.

4.1 Phases

The benchmark is divided into the following phases:

(1) **Initialization**: Loading the transformation and metamodels
(2) **Load**: Loading the input models
(3) **Initial**: Creating the initial Dot file
(4) **Update**: Reading changes to the feature model and propagate

We use the UVL models provided by Sunderman and Feichtinger [9]. For some models, we have change sequences, but unfortunately not for all of them. For those where multiple versions are available, the repository contains diff-files that describe the changes in Git Patch format. If such changes are available, the last step of the benchmark is performed repeatedly. The benchmark seeks to compare execution times for all of the phases. For the last step, any efforts necessary for parsing and loading the changes can be excluded from the time measurements.

4.2 Running the benchmark

The benchmark framework only requires Python 2.7 or above to be installed. Furthermore, the solutions may imply additional frameworks. The reference solution requires .NET 9.0. We ask solution authors to explicitly note dependencies to additional frameworks necessary to run their solutions.

If all prerequisits are fulfilled, the benchmark can be run using Python with the command `python scripts/run.py`. Additional command-line options can be queried using the option `-help`.

```
{
  "Tools": ["NMF"],
  "Models": [{
    "Name": "automotive01",
  }],
  "MaxVersions": 20,
  "Runs": 5,
  "Timeout": 6000
}
```

Listing 2: A minimal benchmark configuration

The benchmark framework can be configured using JSON configuration files. A minimal test configuration is depicted in Listing 2. When creating a new solution, we highly recommend to overwrite the contents of this configuration file locally. In the configuration from Listing 2, only the test scenario with just one minimal model is executed, using only the solution in `Reference` 5 times.

4.3 Input Models

As inputs, we use the UVL models provided by Sunderman and Feichtinger [9]. These are feature models from a variety of different domains.

4.4 Correctness Checks

Correctness of the solutions is currently only evaluated manually, by looking at the resulting Dot files.

4.5 Evaluation Criteria

The solutions are evaluated along the following criteria:

- Correctness
- Understandability
- Conciseness
- Execution time

The understandability of the solutions will be evaluated by a poll during the TTC event. To evaluate the conciseness, we ask every solution to note on the lines of code of their solution. This shall include the model transformation and glue code to actually run the benchmark. Code to convert the change sequence or code generated from the metamodels can be excluded. For any graphical part of the specification, we ask to count the lines of code in a HUTN[3] notation of the underlying model.

In addition, we would like solutions to report whether and to which extend they have used generative AI. If so, we would like solutions to report on the following characteristics:

- Which AI model was used?
- Which prompts were used to generate the solution?
- How much adjustments were necessary?
- How satisfied are you with the code generated by the LLM?

4.6 Solution Requirements

The solutions are required to perform the steps of the benchmark in the order depicted above. Solutions must report the following metrics between these steps, in case of the update phase after every change sequence. The reporting is done by printing the following separated by `;` to the standard output:

- **Tool**: The name of the tool.
- **Model**: The name of the input model set that is currently run
- **RunIndex**: The run index in case the benchmark is repeated
- **Iteration**: The iteration (only required for the Update phase)
- **PhaseName**: The phase of the benchmark
- **MetricName**: The name of the reported metric
- **MetricValue**: The value of the reported metric

Tool, *Model* and *RunIndex* are provided to the solution using environment variables with the same name. Further, the benchmark framework passes the root directory of the models using the variable *ModelDirectory*, the full path of the first model using *ModelPath* and the number of update iterations using *Sequences*.

Solutions should report on the runtime of the respective phase in integer nanoseconds (**Time**) and the working set in bytes (**Memory**). The memory measurement is optional. If it is done, it should report on the used memory after the given phase (or iteration of the update phase) is completed. Solutions are allowed to perform a garbage collection before memory measurement that does not have to be taken into account into the times. In the update phase, we are not interested in the time to parse and identify the changes, but only the pure change propagation.

To enable automatic execution by the benchmark framework, solutions should add a subdirectory to the solutions folder of the benchmark with a *solution.ini* file stating how the solution should be built and how it should be run. Because the solution contains the already compiled reference solution, no action is required for build. However, other solutions may want to run build tools like maven in this case to ensure the benchmark runs with the latest version.

The repetition of executions as defined in the benchmark configuration is done by the benchmark. This means, for 5 runs, the specified command-line will be called 5 times, passing any required information such as the model that should be computed, the run index, etc. in separate environment variables. All runs should all have the same prerequisites. In particular, solutions must not save intermediate data between different runs. Meanwhile, all iterations of the Update phase are executed in the same process and solutions are allowed (and encouraged) to save any intermediate computation results they like, as long as the results are correct after each change sequence.

5 REFERENCE SOLUTION

The repository also contains a reference implementation that mimics how this kind of functionality would be implemented using standard object-oriented code. That is, it uses NMF [4] for the model representation and AnyText [7] to parse UVL files, but otherwise uses simple, standard general-purpose code to directly note down the Dot file.

```
writer.WriteLine("digraph␣FeatureModel␣{");
writer.WriteLine("rankdir=\"TB\"");
writer.WriteLine("newrank=true");
writer.WriteLine("bgcolor=\"#1e1e1e\"");
writer.WriteLine("edge␣[color=white]");
writer.WriteLine("node␣[style=filled␣fontcolor=\"white\"␣fontname=\"Arial␣Unicode␣MS,␣Arial\"];");
writer.WriteLine();

foreach (var feature in featureModel.Features)
{
    WriteFeature(feature, writer);
}
if (featureModel.Constraints.Any())
{
    WriteConstraints(featureModel, writer);
}
writer.WriteLine("}");
```

Listing 3: Snippet from the reference solution

A snippet from the reference solution is depicted in Listing 3. As shown in the listing, the reference solution does not care about the structure of Dot files at all but just outputs plain text. This can

[3]https://www.omg.org/spec/HUTN/

be problematic as it hinders the understandability of the actual transformation that mixes both the actual transformation problem and the concrete syntax of Dot.

6 RELATED WORK

There have been many benchmarks of incremental or bidirectional model processing tools, often originating at the Transformation Tool Contest (TTC). These benchmarks differ from the UVL to Dot benchmark presented in this paper mainly by the relationship between input models and output models.

The *Incremental Lab Workflows Benchmark* [6] investigates correspondences between parts of the input and output that are not 1:1. That is, a single model element in the high-level model may correspond to multiple elements in the target model. This complexity would likely be too much for an LLM, which is why we removed it from the case.

The *Train Benchmark* [10] benchmarks the incremental update performance of solutions for queries on a railway network that detected semantic errors. The benchmark included queries of different complexity targeted at evaluating the incremental pattern matching performance. The change sequences applied were plainly the model update operations to fix the semantic errors detected. Similarly, the *Social Media Benchmark* [5] also aims to benchmark incremental query performance, but on a different domain, with a query that integrated graph algorithms in order to evaluate the integration of custom dynamic algorithms and with change sequences independent from the queries. Both of these benchmarks target plain queries where the structure of the result is completely different than the structure of the input models or not present at all (the result of the social media benchmark is just a set of three pointers to model elements).

In contrast, the *Families to Persons Benchmark* [1] considers a simple bidirectional transformation scenario where input model and output model represent the same information, though according to different metamodels. There is always a clear 1:1 relationship between model elements of either side. The benchmark is thus rather targeted at evaluating and comparing the approaches to support bidirectional transformations. The benchmark does consider incremental updates of the models, but due to the 1:1 mapping, the required efforts to propagate these changes are rather simple and clear.

The *Smart Grid Benchmark* [3] considers queries that merge elements from two models into a view that combines information from both input models. Still, there is a clear 1:1 mapping between tuples of elements of both input models and the output model.

In the *Java Refactoring Benchmark* [8], the task was to extract a simplified refactoring model from a Java code model, then apply refactorings at the simplified model and put back the resulting changes to the Java model. Because the refactoring model featured model elements representing all methods with the same name regardless of declaring class, this transformation also has a more complex mapping than 1:1 relationships. However, the correspondences are not bounded. Furthermore, this benchmark is not equipped with a framework targeting incremental update performance.

In all but the last benchmarks, the models are given in XMI format, which is easy to process by model-riven tools (the kind of tools targetted by the benchmark), but not easy to understand by an LLM. Therefore, we chose to concentrate on a simple DSL (also much easier than Java) as an input model and a textual output format.

REFERENCES

[1] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. 2020. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Softw. Syst. Model.* 19, 3 (2020), 647–691. https://doi.org/10.1007/s10270-019-00752-x

[2] David Benavides, Chico Sundermann, Kevin Feichtinger, José A. Galindo, Rick Rabiser, and Thomas Thüm. 2025. UVL: Feature modelling with the Universal Variability Language. *Journal of Systems and Software* 225 (2025), 112326. https://doi.org/10.1016/j.jss.2024.112326

[3] Georg Hinkel. 2017. The TTC 2017 Outage System Case for Incremental Model Views. In *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017 (CEUR Workshop Proceedings, Vol. 2026)*, Antonio García-Domínguez, Georg Hinkel, and Filip Krikava (Eds.). CEUR-WS.org, 3–12. http://ceur-ws.org/Vol-2026/paper1.pdf

[4] Georg Hinkel. 2018. NMF: A Multi-platform Modeling Framework. In *Theory and Practice of Model Transformation*, Arend Rensink and Jesús Sánchez Cuadrado (Eds.). Springer International Publishing, Cham, 184–194.

[5] Georg Hinkel. 2018. The TTC 2018 Social Media Case. In *Proceedings of the 11th Transformation Tool Contest, co-located with the 2018 Software Technologies: Applications and Foundations, TTCSTAF 2018, Toulouse, France, June 29, 2018 (CEUR Workshop Proceedings, Vol. 2310)*, Antonio García-Domínguez, Georg Hinkel, and Filip Krikava (Eds.). CEUR-WS.org, 39–43. http://ceur-ws.org/Vol-2310/paper5.pdf

[6] Georg Hinkel. 2021. Incremental recompilation of laboratory workflows. In *TTC 2020/2021 - Joint Proceedings of the 13th and 14th Tool Transformation Contests. The TTC pandemic proceedings with CEUR-WS co-located with Software Technologies: Applications and Foundations (STAF 2021), Virtual Event, Bergen, Norway, July 17, 2020 and June 25, 2021 (CEUR Workshop Proceedings, Vol. 3089)*, Artur Boronat, Antonio García-Domínguez, and Georg Hinkel (Eds.). CEUR-WS.org.

[7] Georg Hinkel, Alexander Hert, Niklas Hettler, and Kevin Weinert. 2025. AnyText: Incremental, left-recursive Parsing and Pretty-Printing from a single Grammar Definition with first-class LSP support. In *Proceedings of the 12th International Conference on Software Language Engineering (SLE 2025), June 12-13, 2025, Koblenz, Germany*. ACM. accepted, to appear.

[8] Géza Kulcsár, Sven Peldszus, and Malte Lochau. 2015. Object-oriented Refactoring of Java Programs using Graph Transformation. In *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015 (CEUR Workshop Proceedings, Vol. 1524)*, Louis M. Rose, Tassilo Horn, and Filip Krikava (Eds.). CEUR-WS.org, 53–82. http://ceur-ws.org/Vol-1524/paper3.pdf

[9] Chico Sundermann and Kevin Feichtinger. 2021. *Universal-Variability-Language/uvl-models: SPLC'21 Publication.* https://doi.org/10.5281/zenodo.5031829

[10] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. 2018. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling* 17, 4 (2018), 1365–1393.