# The TTC 2017 Live Contest on Transformation Reuse in the Presence of Multiple Inheritance and Redefinitions

Georg Hinkel
FZI Research Center of Information Technologies
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
hinkel@fzi.de

## Abstract

As model-driven engineering (MDE) gets increasingly applied to larger problems, the modularity of model transformations becomes more and more important. In particular, flexible reuse mechanisms are required to reuse existing transformation logic wherever possible. In this paper, we propose a benchmark for model transformation reuse in the application scenario of generating code for metamodels in the presence of reference refinements and multiple inheritance.

## 1 Introduction

Model transformations are a very important part of any model-driven software development process and have even been called the "heart-and-soul" of this engineering paradigm [1]. As model transformations become more and more complex, their modularity and reuse mechanisms become more important. However, these reuse mechanisms do not seem satisfactory [2], [3].

For a reuse scenario to provide actual advantages, it is further important that the reused artifact encapsulates a substantial share of logic because any kind of reuse always introduces a dependency to the reused artifact. For example, we do not think that reusing a rule that simply copies a name is an artifact worth reusing.

An application scenario where complex logic needs to be shared across multiple model transformations is the code generation for a given metamodel in the presence of refinements (or redefinitions as they are called in the UML [4]) of attributes and references. Such refinements are useful as they allow metamodelers to model relationships in multiple levels of abstraction simultaneously whereas the modeler only has to fill in the values for the least common reference.

However, in a platform that only supports single inheritance of classes such as Java or .NET, refinements step into the decision which existing class can be used as a base class. The logic of this decision making is independent of the way how the features of the class are represented. As outlined in [5], this yields a challenge for the modularity mechanisms of model transformation languages or code generators as such a logic should be made reusable for different purposes of the generated code.

In this paper, we therefore propose this problem as a challenge for authors of model transformation languages to exhibit the reuse mechanisms of their transformation languages. We hope that this challenge and solutions thereof can bring the discussion on reuse in model transformation languages a step forward.

In particular, the goal of this TTC case is to

- compare the mechanisms of state-of-the-art model transformation languages to integrate more complex logic such as in Algorithm 1 and

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

- compare the ability of state-of-the-art model transformation languages to define and reuse skeleton transformations, i.e. transformation stubs that can be reused for multiple concrete model transformations.

Along with this paper, the contest contains online resources[1] that contain input models, expected outputs, a reference solution of this challenge in NTL [6] and a benchmark framework to run solutions, record execution times and generate diagrams.

The remainder of this paper is structured as follows: Section 2 gives more background on the case. Section 3 defines the tasks of this challenge. Section 4 briefly introduces the benchmark framework. Lastly, Section 5 lists the evaluation criteria for solutions of this challenge.

## 2 Case Description

We first explain the code generation in the presence of refinements and then present an algorithm to select a base class for a generated class.

### 2.1 Generating code in the presence of refinements

For any attribute or reference (feature in the remainder), a code generator often generates a property[2] and perhaps other members. If the feature is not refined, this property is usually backed by a field. In case a feature is refined, custom getter and setter implementation are generated that return the value of the refining feature instead.

At the same time, we face the problems that metamodels such as Ecore often allow multiple inheritance but technical platforms such as Java or .NET do not. As a consequence, a class of the metamodel is usually turned into an interface and an implementation class. To minimize the generated code, the code generator often tries to reuse existing implementation classes of base classes in the metamodel.

If an implementation class inherits another implementation class, it inherits all members of this base class. This may include generated members for features that are refined in the scope of the current EClass. Therefore, such an inheritance must be forbidden for the generated code.

Hence, refinements impact the inheritance hierarchy of the implementation base class. In case a feature is refined, the code generator may no longer reuse any implementation class that contains a backing field for this feature.

For example, consider the situation in the left side of Figure 1. There, multiple inheritance is used as class `D` inherits from both `B` and `C` with a common base class `A`. Because the implementation `DImpl` for the metaclass `D` may only use one base class, the property `PropC` has to be replicated in the implementation class `DImpl` as well.[3]

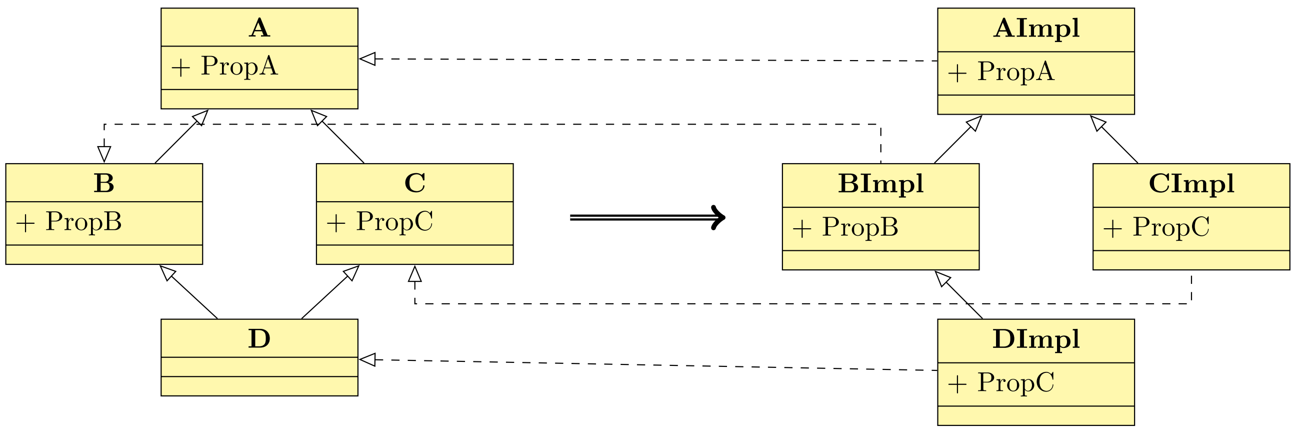

Figure 1: Example resolving multiple inheritance [5]

[1] `http://github.com/georghinkel/ttc2017LiveContest`

[2] A property is a pair of a getter and setter method that have a common name and property type. Properties are well known in a multitude of languages including C# and Python. Java has no explicit language element for properties but properties are sometimes extracted from members heuristically.

[3] Note that in this situation, it is not clear whether `DImpl` should inherit from `BImpl` or `CImpl` and both options would be allowable.

This situation changes if the property `PropB` is a refinement of property `PropA`, as depicted in Figure 2. Now, the implementation class `DImpl` must not use the implementation classes `AImpl` or `CImpl`, because they contain a direct implementation of `PropA`. However, it may use the implementation class `BImpl`.

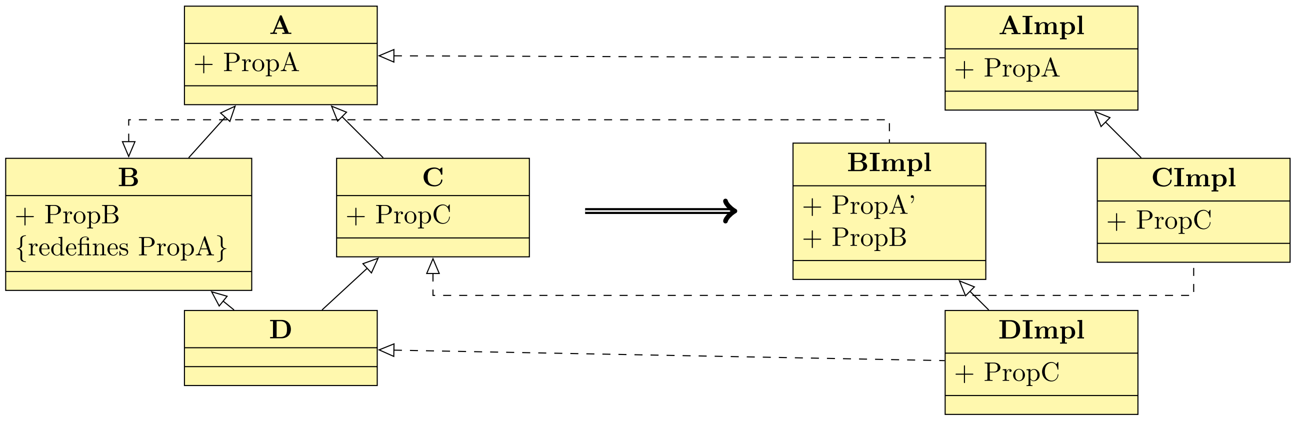

Figure 2: Example resolving multiple inheritance when property `PropB` refines `PropA` [5]

Instead of a full implementation of `PropA`, the implementation class `BImpl` contains a special implementation `PropA'` of `PropA` that instead of backing this property by a field would return the contents of property `PropB`.

As soon as an appropriate base class is found, we may simply copy the generated code for the features that cannot be inherited and copy them into the generated type, as long as they are not refined.

## 2.2 An algorithm to choose the implementation base class

To find such a base class, we propose the abstract algorithm depicted in Algorithm 1. In this algorithm, the relation $\preceq$ denotes reflexive, transitive inheritance, i.e. $c_1 \preceq c_2$ iff $c_2$ is an ancestor of $c_1$. In the examples depicted in Figures 1 and 2, we have that `A` $\preceq$ `A`, `D` $\preceq$ `A` but not `C` $\preceq$ `B`.

Algorithm 1 is essentially based on a reversed topological order of strongly connected components in a dedicated graph that is induced by inheritance relations and refinements. The function Edge implies a graph by deciding whether two nodes are adjacent.

**Algorithm 1** Find implementation base class

```
function AllFeatures($c$) return $\bigcup_{c \preceq c_b} c_b.eStructuralFeatures$
function Refinements($c$) return $\{g | f \in c.eStructuralFeatures, f \xrightarrow{\ll refines \gg} g\}$
function Edge($c_s, c_t$) return $c_s \preceq c_t \vee (\text{Refinements}(c_s) \cap \text{AllFeatures}(c_t) \neq \emptyset \wedge c_t \not\preceq c_s)$
function FindBaseClass($c$)
    $shadows \leftarrow \emptyset$
    $ancestors \leftarrow \text{TransitiveHull}(c, cl \mapsto cl.eSuperTypes)$
    for all $layer$ in ReverseTopologicalOrder($ancestors$, Edge) do
        if $|layer| = 1 \wedge layer \neq \{c\} \wedge shadows \cap \text{AllFeatures}(layer[0]) = \emptyset$ then return $layer[0]$
        for all $l$ in $layer$ do
            $shadows \leftarrow shadows \cup \text{Refinements}(l)$
    return $\perp$
```

In Algorithm 1, the function AllFeatures simply computes the set of all attributes and references available in a given class, including inherited and transitively inherited. The function Refinements returns those attributes and references that are refined by attributes or references of the given class. More interesting is the function Edge that defines the edges in the graph the topological order is created for. This graph shall contain edge from a class $c_s$ to a class $c_t$, if the generated code for class $c_s$ obsoletes the generated code for $c_t$. This may either be because $c_s \preceq c_t$ or because $c_s$ refines a property of $c_t$. The latter case is not problematic for the case that $c_t \preceq c_s$, because in that case, the generated code for $c_t$ is aware of this refinement.

The reversed topological order guarantees us that there is no incoming edge from ancestor classes not yet considered for the given class. It can be easily implemented by reversing the output of Tarjan's algorithm [7].

An implementation of the latter in Java is available in the benchmark resources[4]. In Algorithm 1, we assume the latter to return a list of strongly connected components, each represented as set of classes.

To demonstrate how Algorithm 1 can be implemeted, we explain its implementation in the reference solution in NTL below.

```
private static IEClass FindBaseClass(IEClass eClass, out IEnumerable<IEStructuralFeature> shadows) {
  shadows = Enumerable.Empty<IEStructuralFeature>();
  var ancestors = eClass.Closure(c => c.ESuperTypes);
  foreach (var layer in Layering<IEClass>
         .CreateLayers(eClass, c => Edges(c, ancestors)).Reverse()) {
    if (layer.Count == 1 && layer.First() != eClass &&
       !shadows.IntersectsWith(AllFeatures(layer.First()))) {
      return layer.First();
    }
    foreach (var cl in layer) {
      shadows = shadows.Union(Refinements(cl));
    }
  }
  return null;
}
```

Listing 1: The implementation of the *FindBaseClass* method in the NTL reference solution

Listing 1 shows the implementation of the *FindBaseClass* method. In addition to the class as input, the method has an output parameter to collect all features that have been refined in the scope of the given class. For the computation of the transitive hull, NMF offers a helper method called `Closure`. The implementation for the topological sort is called `Layering` which is reversed using a standard operation on collections. As a parameter, the topological sort implementation takes a method that given a node, selects the adjacent nodes in the graph. In our case, the nodes of the graph are classes and the adjacent nodes are given by the `Edges` function that additionally gets a base set of all classes to consider.

```
private static IEnumerable<IEClass> Edges(IEClass eClass, IEnumerable<IEClass> candidates) {
  var ancestors = eClass.Closure(c => c.ESuperTypes);
  var refinements = Refinements(eClass);
  var conflicting = from cand in candidates
                  where !cand.Closure(c => c.ESuperTypes).Contains(eClass) &&
                     refinements.IntersectsWith(AllFeatures(cand))
                  select cand;

  return ancestors.Union(conflicting);
}
```

Listing 2: The implementation of the *Edges* method in the NTL reference solution

The implementation of this `Edge` function is depicted in Listing 2. It returns all classes $c_t$ for which the `Edge` function in Algorithm 1 returns true when $c_s = eClass$. This includes all ancestors of this class and those that have a feature which is refined in the current scope. This implementation uses a small query and a helper method `AllFeatures` that is depicted below in Listing 3.

```
private static IEnumerable<IEStructuralFeature> AllFeatures(IEClass eClass) {
  return eClass.Closure(c => c.ESuperTypes).SelectMany(c => c.EStructuralFeatures);
}
```

Listing 3: The implementation of the *AllFeatures* method in the NTL reference solution

This method essentially collects all features available for a given class.

## 3 Tasks

We first give an overview on the task, then explain the limitations before we detail on the tasks.

### 3.1 Overview

The task of this challenge is to create two code generators from a version of Ecore that supports refinements to a simplistic object-oriented programming language inspired by C# and Python. To make writing the solutions easier, we created a simple code model for which we provide a printer to a human-readable notation. Therefore,

[4]This implementation uses an edge selection that returns the adjacent nodes connected to a node instead of deciding whether two nodes are adjacent. We included the algorithm to aid the development of solutions, but its usage is not mandatory. Note that the provided implementation of the topological sort is not reversed.

solutions may generate the textual notation or simply implement the code generators as model-to-model transformations that target the metamodel for this language. Both code generators will have to deal with metamodels that contain refinements and multiple inheritance.

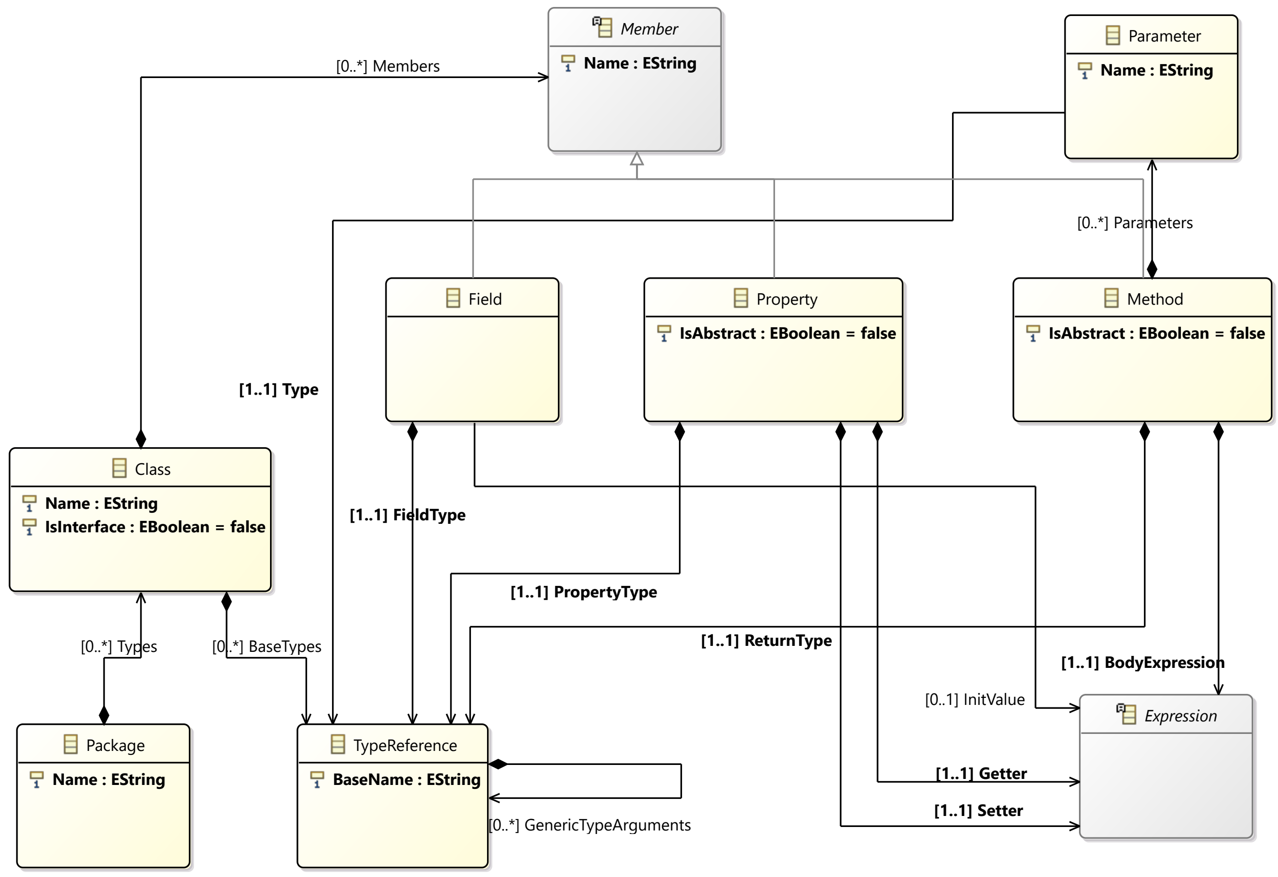

Figure 3: Types and Members in SimpleCodeDOM

This code model is depicted in Figures 3 and 4. An instance of this metamodel is a package. A package may contain classes (that could be interfaces). A class consists of multiple members that can be fields, methods or properties where the latter are simply pairs of a getter and a setter. A class may also define base types (equivalent to super types in Java). Furthermore, there are several expression types.

Methods and properties consist of expressions that represent the code when they are executed. There are multiple types of expressions available, though the case only requires method invocations, binary expressions, conditional expressions, field references (which are also used for property references), string literals, set value references and `this` references.

## 3.2 Limitations

In the scope of this challenge, we only deal with refinement of single-valued references that are refined by exactly one other reference. The example models are guaranteed to be correct in the sense that references only refine references of a base class and that the types of these references fit together.

In particular, the metamodels used for testing only contain instances of the following types:

- `EPackage`: There is exactly one instance per test model.
- `EClass`: There may be multiple classes. Each class may have one or multiple super types.
- `EReference`: Each class may define multiple references to other classes. All references have an upper bound 1. In the scope of this challenge, the information whether a reference is a containment is ignored.

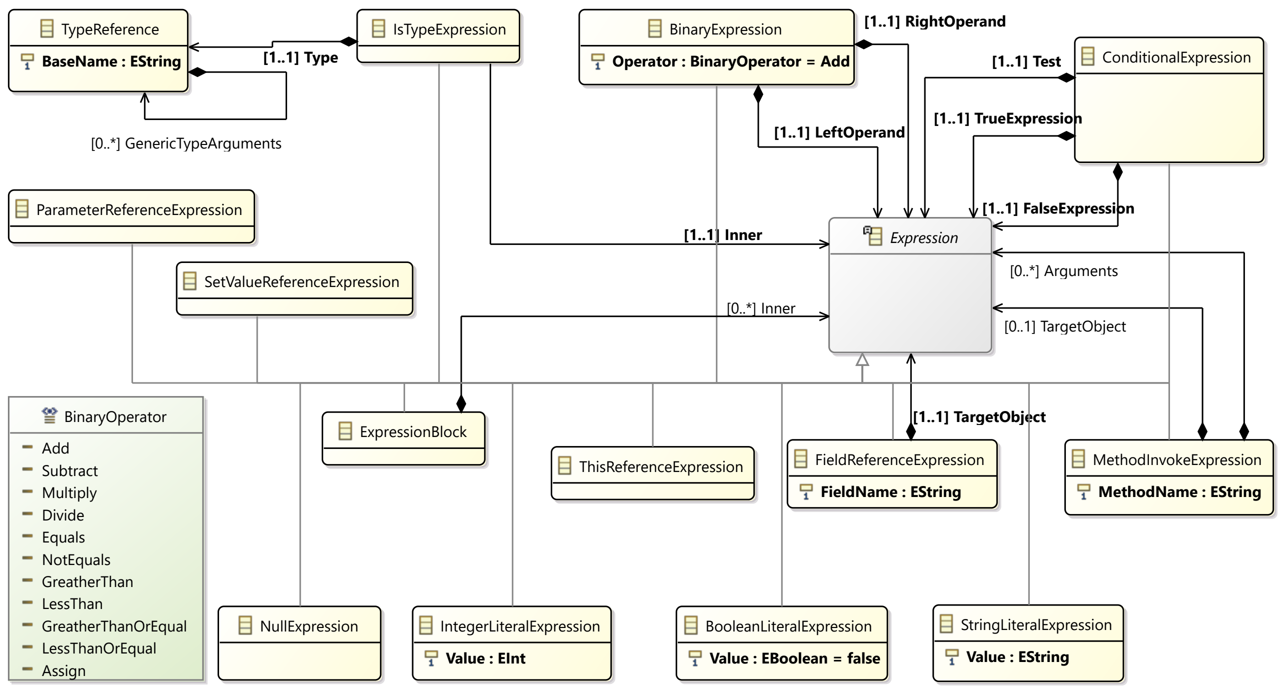

Figure 4: Expressions in SimpleCodeDOM

### 3.3 Code Generator A: Backing references with properties

In the first version of the code generator, a reference should be transformed to a property with a backing field. The property should be named exactly as the original reference. The field should have a prefix "_". The getter and setter of the generated property should simply return or store the backing field. The property for a refined reference should not be backed by a field but should return or store the refining reference.

```
interface A
  abstract property PropA : E

class AImpl : A
  field _PropA : E
  property PropA : E
    get: this._PropA
    set: (this._PropA = value)

interface B : A
  abstract property PropB : E

class BImpl : B
  field _PropB : E
  property PropA : E
    get: this.PropB
    set: (this.PropB = value)

  property PropB : E
    get: this._PropB
    set: (this._PropB = value)
```

Listing 4: The generated code for classes A and B from Listing 2

As an example, we depicted the generated code for the classes A and B from Figure 2 in Listing 4.

Note that we adapted the syntax of C# with respect to inheritance and implementation of interfaces: Both concepts use the same syntax and are not distinguished in the code model where they are both specified as base types. However, by convention, the reference to the base class must be the first.

### 3.4 Code Generator B: Lazy Loading from a database

In a second code generator, we want to implement lazy loading, for example from a database. For this, each reference should be generated as a method that simply invokes an assumed method `resolve` that takes as input an object reference and the name of the reference as a string. The method should have the name of the original reference prepended with "get_". Furthermore, all of the generated classes have to inherit (directly or indirectly) from a common class DBObject.

```
interface A
  abstract method get_PropA()

class AImpl : DBObject, A
  method get_PropA()
    resolve(this, 'PropA')

interface B : A
  abstract method get_PropB()

class BImpl : B
  method get_PropA()
    this.get_PropB()

  method get_PropB()
    resolve(this, 'PropB')
```

Listing 5: The generated code for classes A and B from Listing 1

As an example, we have depicted the expected outcome of this code generator for the model from Figure 2 in Listing 5.

## 4 Benchmark Framework

The benchmark framework is based on the benchmark framework of the TTC 2015 Train Benchmark case [8] and supports a generator of change sequences, automated build and execution of solutions as well the visualization of the results using R. The source code and documentation of the benchmark as well as metamodels, a reference solution in NTL [6], example models, intended outputs for them and a Java implementation of Tarjan's algorithm are publicly available online at http://github.com/georghinkel/ttc2017LiveContest.

### 4.1 Solution requirements

Solutions should report on the runtime of the respective phase in integer nanoseconds (**Time**) and the working set in bytes (**Memory**) by writing the metrics in a CSV format to the standard output. As an example, reporting that the transformation of model model1 using code generator A of the NMF solution for the run index 0 took 83ms and required a working set of 21.7MB, the following lines should be emitted to standard output:

```
NMF;model1;A;0;Time;83761000
NMF;model1;A;0;Memory;22822912
```

Listing 6: Example output

The memory measurement is optional. If it is done, it should report on the used memory after the transformation is completed.

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. In addition, the solution has to be selected in the benchmark configuration file in config/configuration.json.

```
[build]
default=MSBuild NMFCodeGenerator.csproj /p:Configuration=Release
skipTests=MSBuild NMFCodeGenerator.csproj /p:Configuration=Release

[run]
A=bin\NMFCodeGenerator.exe transformA
B=bin\NMFCodeGenerator.exe transformB
```

Listing 7: An example solution.ini file

As an example, Listing 7 contains solution.ini file for the NMF reference solution. It simply states the executable that should be run and the transformation that should be called. All other parameters such as the model input and output path are passed through environment variables Input, Output, Model and RunIndex.

### 4.2 Running the benchmark

The benchmark framework only requires Python 2.7 or above and R to be installed. R is required to create diagrams for the benchmark results[5]. Furthermore, the solutions may imply additional frameworks. We would 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": [
    "model1",
    "model2",
    "model3",
    "model4"
  ],
  "Transformations": ["A", "B"],
  "Runs": 5
}
```

Listing 8: The default benchmark configuration

The benchmark framework can be configured using JSON configuration files. The default configuration is depicted in Listing 8. In this configuration, code for all example models is computed using only the reference solution in NMF 5 times each.

To execute the chosen configuration, the benchmark can be run using the command line depicted in Listing 9.

```
&> python scripts/run.py
```

Listing 9: Running the benchmark

Additional commandline parameters are available to only update the measurements or create visualizations.

## 5 Evaluation

Solutions of the proposed benchmark should be evaluated by their completeness, correctness, conciseness, understandability, percentage of shared code and performance.

For each evaluation category, a solution can earn 5 points.

### 5.1 Completeness & Correctness

Assessing the completeness and correctness of model transformations is a difficult task. In the scope of this benchmark, completeness and correctness are checked in terms of the example models: For each model, we make a string comparison of the printed code.

Points are awarded proportional to the test cases that match exactly. If all tests pass, 5 points are awarded.

The benchmark framework performs a string comparison for the generated code or the print of the generated model using our printer.

### 5.2 Conciseness

As for any model transformation tasks, conciseness is important to make sure the implementation is of a reasonable size.

To evaluate the conciseness, we ask every solution to note on the lines of code of their solution. For any graphical part of the specification, we ask to count the lines of code in a HUTN[6] notation of the underlying model.

The points in this category are awarded according to the following formula:

$$Points = 5 \cdot \frac{\log Loc_{leastconcise} - \log LoC}{\log Loc_{leastconcise} - \log LoC_{mostconcise}}.$$

[5]More details on how to run the benchmark framework can be obtained at the benchmark repository.

[6]`http://www.omg.org/spec/HUTN/`

### 5.3 Understandability

Similarly to conciseness, it is important for maintainance tasks that the solution is understandable. However, as there is no appropriate metric for understandability available, the assessment of the understandability is done manually. For solutions participating in the contest, this score is collected using questionnaires at the workshop.

### 5.4 Share of reused code

In this category, we evaluate the percentage of model transformation code that is shared. Points are awarded proportional to the following formula:

$$\frac{\text{Lines of code shared}}{\text{Total Lines of code for code generator A}} + \frac{\text{Lines of code shared}}{\text{Total Lines of code for code generator B}}.$$

### 5.5 Performance

For the performance measurements, we measure the execution time for some example models. Points are awarded according to the following formula:

$$Points = 5 \cdot \frac{\log Time_{slowest} - \log Time}{\log Time_{slowest} - \log Time_{fastest}}.$$

The recording of execution times has to be done by each solution separately. The time measurement should contain loading the input models, the actual transformation and serializing the output models.

### 5.6 Overall Evaluation

Due to their importance, the points awarded in completeness & correctness and understandability are doubled in the overall evaluation.

## References

[1] S. Sendall and W. Kozaczynski, "Model transformation the heart and soul of model-driven software development," Tech. Rep., 2003.

[2] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Fact or fiction–reuse in rule-based model-to-model transformation languages," in *Theory and Practice of Model Transformations*, Springer, 2012, pp. 280–295.

[3] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: Are we there yet?" *Software & Systems Modeling*, pp. 1–36, 2013.

[4] Object Management Group (OMG), *Unified Modeling Language (UML) – Version 2.5 (formal/2015-03-01)*, 2015.

[5] G. Hinkel, T. Goldschmidt, E. Burger, and R. Reussner, "Using Internal Domain-Specific Languages to inherit Tool Support and Modularity for Model Transformations," *Software & Systems Modeling*, pp. 1–27, 2017.

[6] G. Hinkel, "An approach to maintainable model transformations using an internal DSL," Master's thesis, Karlsruhe Institute of Technology, 2013.

[7] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[8] G. Szárnyas, O. Semeráth, I. Ráth, and D. Varró, "The TTC 2015 train benchmark case for incremental model validation," 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.*, 2015, pp. 129–141.