

An NMF solution for the *Flow Graphs* case study at the TTC 2013

Georg Hinkel
Karlsruhe Institute of Technology
Karlsruhe, Germany
georg.hinkel@student.kit.edu

Thomas Goldschmidt
ABB Corporate Research
Ladenburg, Germany
thomas.goldschmidt@de.abb.com

Lucia Happe
Karlsruhe Institute of Technology
Karlsruhe, Germany
lucia.kapova@kit.edu

Software systems are getting more and more complex. Model-driven engineering (MDE) offers ways to handle such increased complexity by lifting development to a higher level of abstraction. A key part in MDE are transformations that transform any given model into another. These transformations are used to generate all kinds of software artifacts from models. However, there is little consensus about the transformation tools. Thus, the Transformation Tool Contest (TTC) 2013 aims to compare different transformation engines. This is achieved through three different cases that have to be tackled. One of these cases is the Flowgraphs case. A solution has to transform a Java code model into a simplified version and has to derive control and data flow. This paper presents the solution for this case using NMF Transformations as transformation engine.

1 Introduction

The challenge of the Flowgraphs Case [3] is to derive a control flow graph and data flow graph from a JaMoPP [2] model representing Java code. The case is divided into four subtasks. The first task deals with the creation of an initial flow graph model out of the JaMoPP model representing the Java program. The second task is to derive the control flow within this flow graph model. The third task deals with the issue to derive the data flow out of the prior. Task four finally demands a tool to validate the output of the previous transformations. The transformation tasks are tackled with NMF¹, an open source project to support model-driven engineering on the .NET platform. The solution is available on SHARE².

2 .NET Modelling Framework (NMF)

The .NET Modelling Framework is an open source project that provides support for model-driven software development on the .NET platform. An essential part is the model transformation engine, NMF TRANSFORMATIONS, which allows to write rule-based transformations in arbitrary .NET languages using an internal DSL [1]. The reason to implement the transformation language as internal DSL is mainly that transformation languages ought to be Turing complete [4] and thus, many advantages of external DSLs attenuate. An internal DSL, however, can make use of features of its host language. Developers used to this language feel familiar with the DSL.

NMF TRANSFORMATIONS makes it possible to specify model transformations directly in C#. For this purpose, NMF TRANSFORMATIONS has a simple abstract syntax but hides the complexity in the

¹<http://nmf.codeplex.com/>

²http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC13::NMF_TTC13::NMF_updated_NMF_LiveContest.vdi

attributes of the metaclasses which are representing functions. These functions can be specified with general purpose code that contain code as sophisticated as required. Although the transformation language might seem quite verbose, especially when compared with external model transformation languages, C# has been chosen as host language to make it easier to write and thus maintain these transformations for C# developers.

Currently, NMF does not contain a metamodeling foundation, e.g. based on MOF. Instead, NMF TRANSFORMATIONS uses the concepts of the CLR (the virtual machine used on the .NET platform) to represent models and operates on plain objects (POCOs). Thus, we used an interop component to EMF, which generates classes from an Ecore metamodel. Furthermore, there exists a serializer component to load and store simple models (models that do not have references to other files).

Beside NMF, to the best of our knowledge, there is hardly any framework that supports model-driven engineering on the .NET platform. Microsoft offers a Visualization and Modeling SDK³ for Visual Studio, which is a tool for graphical editors, and T4 as Text-To-Text-Transformation engine. However, although T4 is included in Visual for many years now, there is still hardly support for editing T4 templates. There is an add-in providing syntax-highlighting and code-completion, but still the support is much less than for writing normal e.g. C# code. Furthermore, T4 has some restrictions like no inheritance is allowed within a T4-template. These restrictions and the lack of out-of-the-box tool support make model-driven development hard. NMF TRANSFORMATIONS makes it possible to use the full tool support for C# also for model transformations.

3 Solution

All of the tasks have been tackled. The final solution consists of two separate C# console projects. The first console application reads a JaMoPP file, transforms it to a Structure Graph (Task 1, see section 3.1), derives the control flow (Task 2, see section 3.2) and sets the data flow (Task 3.2, see section 3.3). Each of the transformations operates in memory, only. After all transformations have been applied, the resulting data flow model is persisted using XMI into an output file specified by command line parameters. Furthermore, deriving the control flow graph and setting the data flow can be switched on or off using command line arguments. The second console application validates the links (see section 3.4).

3.1 Task 1/3.1: Structure Graph

A M2M-transformation in NMF TRANSFORMATIONS is specified through transformation rules, which are represented by classes. These transformation rules may only be called once per input arguments in a transformation context. This context represents a transformation pass and provides trace functionality. Furthermore, transformation rules may define dependencies to other rules. These dependencies are necessary to set, in order to let NMF Transformations know which rules to call and to derive the inputs of these rules. NMF TRANSFORMATIONS operates on plain CLR objects and therefore does not know the structure of the metamodel. Thus the structure of the domain model has to be reflected in the dependencies of the transformation rules. The only rule that is actually called by NMF TRANSFORMATIONS automatically is the rule that matches the transformation request to transform a Java code model into a flow graph.

Due to space limitations, only an example of the transformation rules involved in the initialization can be shown here. Figure 1 shows the transformation of *AssignmentExpression* elements. The code within

³<http://archive.msdn.microsoft.com/vsvmsdk>

```

public class AssignmentExpression2Text : TransformationRule<JaMoPP.Expressions.AssignmentExpression, string>
{
    public override string CreateOutput
        (JaMoPP.Expressions.AssignmentExpression input, ITransformationContext context)
    {
        var child = context.Trace.ResolveIn
            (Rule<Expression2Text>(), input.Child);
        var value = context.Trace.ResolveIn
            (Rule<Expression2Text>(), input.Value);
        var assignment = context.Trace.ResolveIn
            (Rule<AssignmentOperator2Text>(), input.AssignmentOperator);

        return child + assignment + value;
    }

    public override void RegisterDependencies()
    {
        MarkInstantiatingFor(Rule<Expression2Text>());

        Require(Rule<Expression2Text>(), selector: expr => expr.Child);
        Require(Rule<Expression2Text>(), selector: expr => expr.Value);
        Require(Rule<AssignmentOperator2Text>(), selector: expr => expr.AssignmentOperator);
    }
}

```

Figure 1: The transformation of assignment expressions

RegisterRequirements shows how dependencies are specified using lambda expressions that define which subsequent elements to call. This correspondence can be queried later on. As the target model for expressions is plain strings, which are immutable in .NET, the transformation efforts need to be done within *CreateTransformationOutput*.

However, as Task 3.1 requires to also set the definitions of variables, unlike the initialization from Task 1, Task 3.1 transforms expressions into more complex objects that inherit from an interface to construct the Expr elements and set the *def* and *use* links accordingly. This interface is presented in Figure 2.

```

public interface IExpressionDFInfo
{
    string Expression { get; }

    IEnumerable<Flowgraph.Var> UsedVariables { get; }

    Flowgraph.Var LastVariable { get; }

    void SetDefs(Flowgraph.FlowInstr parentFlow);
}

```

Figure 2: The interesting attributes for an Expression

By using such mutable objects, dependencies may also specify how they are persisted in the output of the transformation rule via persistors.

The *MarkInstantiatingFor* method marks the assignment transformation rule as instantiating for the *Expression2Text*-rule, i.e. whenever an *Expression* should be transformed to text and the *Expression* is an *AssignmentExpression*, *AssignmentExpression2Text* is called instead to create the output of the *Expression2Text*-rule. *Expression2Text* is still called, but here it is empty and serves as a hub, only. Thus, the transformation of an expressions to text does not need to know which concrete expressions

exist. The *Require* methods specify dependencies to other model elements.

3.2 Task 2: Control Flow Graph

To derive the control flow graph, semantical information such as the first flow instruction within a statement has to be added to existing model elements. But it is not only data, but also the behaviour of how to set the control flow, that is important for this transformation. Thus, we define the necessary operations on statements that are necessary to derive the control flow, see Figure 3.

```
public interface IControlFlowInformation
{
    FlowInstr First { get; }
    FlowInstr Successor { get; set; }
    void SetControlFlow(Stack<Stmt> callHierarchy, ITransformationContext context);
}
```

Figure 3: The interface of what is interesting regarding control flow

At first, we need the first flow instruction since the entrance in a statement (*First*). However, in some cases like empty blocks, this first flow instruction is not part of the current statement. Therefore, we need to tell what the next flow instruction after the current statement is (*Successor*). Finally, the procedure of how to set the control flow for a statement also depends on a statement. A simple statement only sets the CfNext reference to the first inner flow instruction of the successor statement, whereas a jump statement sets a CfNext reference to the target jump label. However, some statements like `break` and `continue` cannot set their control flow successor without context. I.e., the successor of a `continue` statement is the test expression of the innermost loop that the `continue` is contained in. Additionally, the method also has the transformation context as parameter for tracing purposes.

NMF TRANSFORMATIONS does not draw a difference between objects that are part of the model and objects that are just helpers. Thus, we can just use transformation rule that return implementations of the above interface for any statement. In an in-place transformation rule we only need to execute the resulting *SetControlFlow* method for the *Method* element.

3.3 Task 3.2: Deriving Data Flow

Deriving the data flow has been implemented in general purpose code. As NMF TRANSFORMATIONS operates on POCOs, integrating this general purpose code in the transformation is just as easy as calling the algorithm. There is no conversion that has to be done. The algorithm works in that way that for a given variable, it follows the control flow everywhere until it arrives at a flow element that defines the same variable and thus the definition loses scope.

3.4 Task 4: Validation

As the proposed query language is very simple, it suffices to solve the problem with regular expressions. To parse a command, we use the following regular expression:

```
(?<command>(cfNext|cfPrev|dfNext)):\s*
" (?<source>[~"]*) "\s*-->\s*" (?<target>[~"]*) "(;)?
```

The validation application now just reads a target model and creates an internal hashtable with all the instructions that are contained in the model. Any validation string is then parsed with the above pattern and the application simply checks whether the asserted condition holds.

4 Validation

So far, the results have only been analyzed by reviewing the results for the output XMI files. All of the output XMI files validated successfully in Eclipse.

The execution times on SHARE transforming the simple input models were really fast, see Table 1. The execution times for the smaller models were measured using hardware query performance counters for more exact time measurement. Otherwise execution times like these would be unable to measure.

The performance results show that the simplified algorithm to derive the data flow maybe is not that fast and possibly needs improvement. However, as this sort of algorithm was written in general purpose code, further optimization is not in the scope of this paper.

5 Conclusion

In this paper we have presented a solution to the TTC 2013 *Flowgraphs* case based on NMF TRANSFORMATIONS. It was not possible to support every bit of the transformation with NMF, as no support for iterative procedure is offered. However, it was easy to integrate general purpose code and cooperate with it.

We suggest the high points of our solution as

- **Good maintainability** through small change impacts as new metamodel elements are introduced
- **Excellent execution speed** with hardly measurable execution times except for the biggest models
- **Easy integration of general purpose code** whenever a task cannot be supported by NMF directly and has to be accomplished with general purpose code
- **Great tool support**, as NMF TRANSFORMATIONS can reuse for example debugging, profiling, refactoring, testing and continuous integration support for C#.

The extension of the transformation from the initial solution to the updated solution supporting also unary expressions also showed that transformations in NMF TRANSFORMATIONS are easy to maintain, as no existing code had to be changed to support this new requirement.

References

- [1] Martin Fowler (2010): *Domain-specific languages*. Addison-Wesley Professional.
- [2] Florian Heidenreich, Jendrik Johannes, Mirko Seifert & Christian Wende (2009): *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik.
- [3] Tassilo Horn (2013): *The TTC 2013 Flowgraphs Case*. http://planet-sl.org/ttc2013/images/userdirs/618/ttc2013_flowgraphs_case.pdf.
- [4] Shane Sendall & Wojtek Kozaczynski (2003): *Model transformation: The heart and soul of model-driven software development*. *Software, IEEE* 20(5), pp. 42–45, doi:10.1109/MS.2003.1231150.

A Appendix

Test case	Reading input	Transformation	Derive Control Flow	Derive Data Flow	Writing output
0	48.90ms	4.48ms	0.29ms	0.08ms	9.63ms
1	38.62ms	4.45ms	1.50ms	0.09ms	4.76ms
2	34.65ms	4.35ms	0.72ms	0.11ms	5.24ms
3	46.03ms	3.01ms	0.48ms	0.05ms	4.87ms
4	36.32ms	2.72ms	0.47ms	0.04ms	4.63ms
5	31.33ms	3.71ms	1.05ms	0.05ms	4.13ms
6	32.77ms	4.60ms	0.56ms	0.04ms	4.21ms
7	168.36ms	27.15ms	5.54ms	5.73ms	26.79ms
8	532.42ms	97.08ms	30.16ms	68.26ms	97.02ms
9	3,832.02ms	1,103.49ms	350.14ms	6,345.40ms	646.39ms
10	33.30ms	1.94ms	0.41ms	0.03ms	4.20ms
11	29.33ms	3.36ms	0.34ms	0.05ms	3.90ms

Table 1: Execution times of the test cases