Using Meta-code Generation to Realize Higher-order Model Transformations

Thomas Buchmann and Felix Schwägerl
Chair of Applied Computer Science I, University of Bayreuth, Universitaetsstrasse 30, 95440 Bayreuth, Bayreuth, Germany

Keywords: Model-driven Development, M2M Transformations, Model Transformations, Higher-order Transformations, Software Product Lines.

Abstract: Model-driven engineering is a wide-spread paradigm in modern software engineering. During the last couple of years, many tools and languages have been developed, which are especially designed for model transformations — a discipline which is needed in many model-driven engineering approaches. While most of the existing model-to-model tools and languages are tailored towards batch transformations for specific model instances, they lack support for generic transformation problems, where the metamodel is unknown beforehand. In this paper we present a two-step meta-code generation approach that derives a metamodel-specific model-to-model transformation from a model-to-text transformation. The approach has been successfully applied to the problem of product derivation in model-driven software product lines.

1 INTRODUCTION

Model-driven engineering (MDE) is a wide-spread paradigm in modern software engineering. It puts strong emphasis on the development of higher-level models rather than on source code. Many MDE disciplines are supported by *model transformations*. A wide range of languages and tools have been developed (Czarnecki and Helsen, 2006), including e.g. QVT (OMG, 2011) and ATL (Jouault et al., 2008) for model-to-model transformations (M2M) or the MOF *model-to-text* standard (OMG, 2008) for model-to-text transformations (M2T). At present, the technology for defining and executing unidirectional batch transformations seems to be well-developed.

Usually the metamodels for which a model transformation is written are known beforehand. As classified in (Mens and van Gorp, 2006), model transformations can be *exogenous*, i.e. source and target models are instances of different metamodels, or *endogenous*, i.e. both models conform to the same metamodel. The terms *in-place* and *out-place* refer to transformations which do or do not modify the source model.

While common approaches work fairly well for fixed metamodels, they lack support for mechanisms such as reflection, which could be used in cases where only the meta-metamodel is known during development time. Only recently, the concept of *higher-order transformations* (HOT) has been introduced as a possible compensation for that lack: A HOT is a transformation that produces or modifies another executable transformation. Unfortunately, there is no mature tool support for higher-order transformations yet.

In this paper, we present a two-step *meta-code generation* approach to compensate that lack. We use an Acceleo[1] model-to-text transformation to generate ATL[2] code for an endogenous out-place model transformation at runtime. We apply our approach to a specific problem scenario — the derivation of products in model-driven software product lines. Nevertheless, the approach can be transferred to many other common MDE problems.

2 PROBLEM DESCRIPTION

One area of our research is dedicated to model-driven *software product line engineering* (SPLE). SPLE (Clements and Northrop, 2001) addresses the organized reuse of software artifacts. Feature models (Kang et al., 1990) are used to capture the commonalities and differences of members of a product line, while feature configurations describe the characteris-

[1] `http://www.eclipse.org/acceleo`; An implementation of the OMG standard *model to text*; (OMG, 2008)

[2] `http://www.eclipse.org/atl`; A model-to-model transformation language for EMF (Jouault et al., 2008)

Buchmann T. and Schwägerl F..
Using Meta-code Generation to Realize Higher-order Model Transformations.
DOI: 10.5220/0004522305360541
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-PT-2013), pages 536-541
ISBN: 978-989-8565-68-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

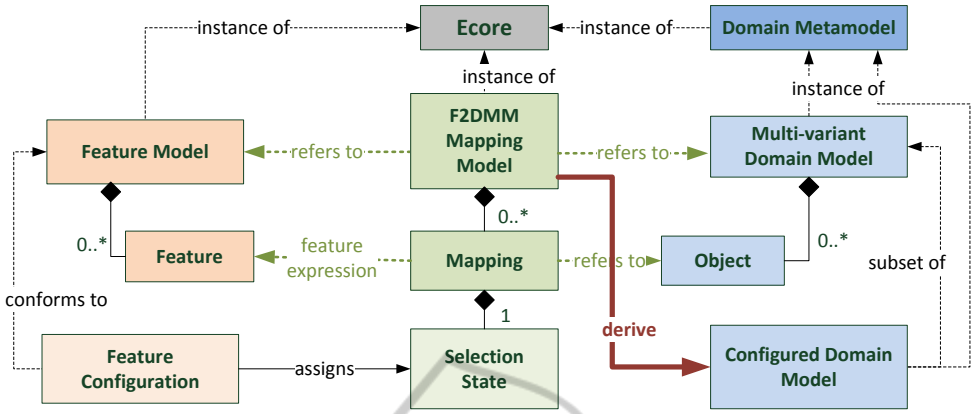

Figure 1: Conceptual overview on the FAMILE toolchain.

tics of a specific member thereof. Software product line engineering is divided into two levels. (1) *Domain engineering* is used to analyze the domain and capture the commonalities and variabilities in a feature model. Furthermore, the features are realized in a corresponding implementation. In model-driven software product lines, models represent the implementation at a higher level of abstraction. (2) *Application engineering* deals with binding the variability defined in the feature model and deriving concrete products.

In SPLE, basically two different approaches exist to realize variability in the corresponding feature implementation: (1) In approaches based on *positive variability*, product-specific artifacts are built around a common core. During application engineering, composition techniques are used to assemble the final product using these artifacts. (2) In approaches based on *negative variability*, a superimposition of all variants is created. The derivation of products is achieved by removing all fragments of artifacts implementing features which are not contained in the specific feature configuration for the desired product.

Several approaches exist to associate elements of the feature model with artifacts part of the domain model; in previous publications (Buchmann and Schwägerl, 2012) we presented FAMILE, an EMF-based toolchain for model-driven software product line development using negative variability. Figure 1 depicts the different models involved in FAMILE. The left part shows the feature model and corresponding configurations. The right part of the figure depicts the *multi-variant* domain model – the superimposition – which contains the implementation of all features described in the feature model. As stated above, FAMILE uses negative variability, i.e. the domain model is the superimposition of all variants. The central part of Figure 1 depicts the *feature-to-domain mapping model* (F2DMM). It is used to map features declared in the feature model to the corresponding implementation fragments in the domain model. *Feature expressions*, which can be arbitrary logical combinations of features, are used for that purpose. For a selected feature configuration, all feature expressions assigned in the mapping model are evaluated to determine the *selection state*s of mappings. Selection states are used to include or exclude corresponding domain model elements in/from a configured domain model.

Product derivation comprises the creation of a configured domain model by filtering all domain model elements which refer to feature expressions evaluating to false w.r.t a given feature configuration. It can be regarded as an *endogenous out-place* (Mens and van Gorp, 2006) model-to-model transformation: Source and target model are both instances of the same meta-model, but the respective instances differ. In our particular case, the target model is a subset of the source model, which means that the problem can be solved by a *conditional copying* transformation: only elements mapped to positive selection states (i.e. *active*, *enforced* or *surrogated*) are copied from the source to the target model; details see (Buchmann and Schwägerl, 2012).

The initial implementation of this transformation was realized by a hand-written model-to-model transformation specified in Java. The transformation redefined the EMF copy operation (`EcoreUtil.copy(EObject)`) which uses the reflective EMF layer to instantiate the copied object from the respective `EClass` of its original. Using this mechanism, the concrete `EFactory` is called at runtime to produce an instance of the desired class. Furthermore, its structural features are traversed and also copied if implied by their respective selection states.

As we encourage the use of model-driven techniques in our research, we strived for a more declara-

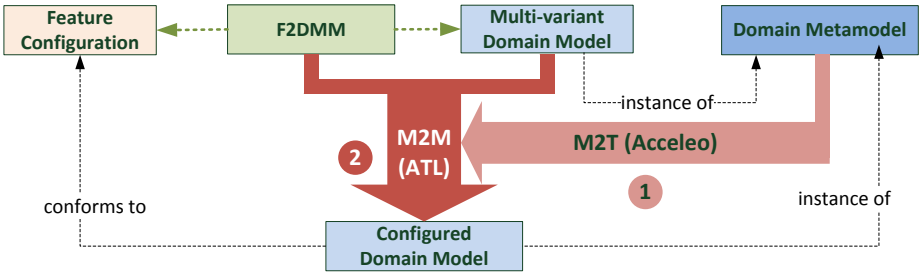

Figure 2: Transformation steps included in our product derivation process.

tive solution to solve this task. Using a M2M transformation language like ATL, copying an object of a specific class could be implemented by a simple transformation rule including a *guard* which ensures that the rule is applied only if the associated mapping has a positive selection state. The rule itself copies the values of all structural features. FAMILE and F2DMM in particular allow the use of any Ecore-based domain model, but ATL and other model-to-model transformation languages require the specification of concrete metamodels. Consequently, this problem can only be solved by means of a generic M2M transformation.

3 META-CODE GENERATION

If ATL provided native support for *generic* transformations, we could write down a single transformation rule to match all classes `C` from the domain metamodel (cf. Listing 1). As stated in Section 2, deriving concrete products from the multi-variant domain model can be regarded as a model transformation realizing a conditional copy operation.

Listing 1: Fictional ATL rule augmented with generics and reflection.

```
1  rule<C extends EObject> Object2Object {
2    from
3      source : C ( source.isIncluded() and
4                   source.eClass = C )
5    to
6      target : C (
7        for (EStructuralFeature feature :
8                C.eAllStructuralFeatures) {
9            feature <- source.feature
10       }
11     )
12 }
```

For our generic transformation rule above, we augmented the ATL syntax with fictional constructs based on Java generics and OCL queries for accessing the metamodel reflectively. The type parameter `C` attached to the `Object2Object` rule is lower-bounded by the basic EMF Object type `EObject`. The guard consists of a conjunction of two queries: `source.eClass = C` ensures that no instances of sub-classes of `C` are matched to the rule and `source.isIncluded()` requires that an object's associated feature expression evaluates to a positive selection state. Inside the rule body, the `for` loop accesses the reflective EMF layer to iterate over the structural features of the class bound to the type parameter at runtime. Please note that in general purpose programming languages like Java or Ada, a generic type is statically instantiated during compile time. In our case, the type parameter has to be bound dynamically depending on the type of the source object. Unfortunately such mechanisms do neither exist in ATL nor in other common M2M languages. In the following we describe how we applied our two-step *meta-code generation* approach (cf. Figure 2) to simulate such a generic behavior.

In the first step we execute an Acceleo model-to-text transformation that takes as input the metamodel of the multi-variant domain model. At runtime, it can be retrieved from the current F2DMM instance, as the mapping model stores a reference to the domain model (cf. Figure 1). Using EMF's reflective layer, the meta-class for each `EObject` and thus the containing `EPackage` can be retrieved. Afterwards, the M2T transformation is run on that `EPackage`. The template shown in Figure 3 produces four kinds of ATL code fragments:

- The *header* (lines 7 to 11): The metamodel identifier and the namespace URI of the domain metamodel as well as the module name are produced by the transformation at runtime, while the corresponding information related to the F2DMM metamodel is fixed.
- *OCL-helpers* (lines 13 to 19). They are produced for every class contained in the respective `EPackage` of the domain metamodel which represent a root in the inheritance hierarchy. The generated code is a number of `isIncluded()`-queries containing OCL expressions that find the mapping for an instance of the given class and check if its

```
1  [comment encoding = UTF-8 /]
2  [module generate('http://www.eclipse.org/emf/2002/Ecore')]
3
4  [comment @main /]
5  [template public generateElement(pkg : EPackage)]
6  [file (pkg.atlFileName(), false, 'UTF-8')]
7      -- @nsURI [pkg.name.toUpper()/]=[pkg.nsURI/]
8      -- @nsURI F2DMM=http://ai1.inf.uni-bayreuth.de/f2dmm/1.0.0
9
10 module [pkg.name/]2[pkg.name/];
11 create OUT : [pkg.name.toUpper()/] from IN : [pkg.name.toUpper()/], F2D : F2DMM;
12
13 [for (ec: EClass | pkg.getAllClasses()->select(cl | cl.isRoot()) )]
14 [if (ec.isContainedInModelInstance(true))]
15 helper context [pkg.name.toUpper()/]![ec.name/] def: isIncluded() : Boolean =
16     Sequence{#active, #enforced, #surrogated}
17     ->includes(F2DMM!MappingModel.allInstances
18         ->first().getObjectMapping(self).selectionState);
19 [/if][/for]
20
21 [for (ec: EClass | pkg.getAllClasses()->reject(cl | cl.abstract) )]
22 [if (ec.isContainedInModelInstance(false))]
23 rule [ec.name/]2[ec.name/] {
24     from
25         source : [pkg.name.toUpper()/]![ec.name/] (source.isIncluded() and
26                 source.oclIsTypeOf([pkg.name.toUpper()/]![ec.name/]))
27     to
28         target : [pkg.name.toUpper()/]![ec.name/] (
29             [for (sf : EStructuralFeature | ec.eAllStructuralFeatures
30                 ->select(sf | sf.isPersistent())) separator(', ')]
31             [if (sf.isAttribute())]
32                 [sf.name/] <- source.[sf.name/]
33             [elseif (sf.isContainment())]
34                 [sf.name/] <- source.[sf.name/]
35             [elseif (sf.isCrossref())]
36                 [sf.name/] <- source.[sf.name/]
37             [/if]
38         [/for]
39     )
40 }
41 [/if][/for][/file][/template]
```

Figure 3: Cut-out of our M2T transformation which produces a metamodel-specific ATL file.

associated selection state implies its inclusion (*active*, *enforced* or *surrogated*) in the configured domain model.

- *Conditional copy rules* (lines 21 to 41): For each concrete class from the input domain metamodel, a copy transformation rule is produced which uses the appropriate `isIncluded()`-query as a guard and copies instances of that class. Similar to Listing 1, the produced rule body will copy the attributes and references of the given object.

In order to avoid the creation of unnecessary ATL code, we additionally perform a check if at least one concrete instance of a meta-class is actually contained in the multi-variant domain model (`isContainedInModelInstance()`). Only if this query evaluates to true, a corresponding transformation rule is generated. This results in a more compact and faster ATL transformation.

The output of the first (higher-order) transformation step is an ATL file that describes a (first-order) transformation which has both the multi-variant domain model and the F2DMM mapping model as input. This transformation produces the configured domain model, i.e. the subset of the multi-variant domain model which conforms to the selected feature configuration.

As an example, a cut-out of the generated ATL module file for the Eclipse UML2 metamodel is shown in Listing 2. The ATL transformation uses instances of the UML2 as well as the F2DMM metamodel as input to create a filtered UML model as output. The *conditional copy* behavior is realized as follows: The rule `Class2Class` is used to create new `Classes` in the configured domain model. The guard `source.isIncluded()` next to the source pattern makes use of the helper function defined above. It is only produced for the `UML!Element` which is the base class of all metaclasses in the UML package.

Please note that the derivation process is fully automated and both transformations described in this section are performed “under the hood”: The user can invoke the complete meta-code generation with a single click. The framework then performs both transformations in order to produce the configured domain model. Optionally, the user can choose to serialize the derived ATL transformation which can be reused for different feature configurations.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Listing 2: Cut-out of the generated ATL file for the Eclipse UML2 metamodel.

```
1  module uml2uml;
2  create OUT : UML from IN : UML, F2D : F2DMM;
3
4  helper context UML!Element
5      def: isIncluded() : Boolean =
6    Sequence{#active, #enforced, #surrogated}
7    ->includes(F2DMM!MappingModel.allInstances
8      ->first().getObjectMapping(self)
9      .selectionState);
10
11 rule Class2Class {
12   from
13     source : UML!Class ( source.isIncluded()
14       and source.oclIsTypeOf(UML!Class) )
15   to
16     target : UML!Class (
17       eAnnotations <- source.eAnnotations
18 ,       ownedComment <- source.ownedComment
19 ,       name <- source.name
20       // ...
21     )
22 }
```

4 RELATED WORK

As stated in Section 1, common model transformation approaches lack support for higher-order tranformations. However, there are some experimental extensions available for some of those tools. In this section we briefly compare them to our approach.

EMF Henshin (Arendt et al., 2010) is a model transformation tool which allows to specify both endogenous or exogenous model transformations using a graphical syntax. The tool is based upon the concept of graph transformations. In (Biermann et al., 2010), the authors present a solution for an `Ecore2GenModel` transformation which also makes use of higher-order transformations. However, there are significant differences to our approach. For example, both metamodels (the Ecore metamodel and the Ecore generator metamodel) are known when the higher-order transformation is specified. Furthermore, the higher-order transformation specified in the approach of Biermann et al. requires the modeler to precisely be familiar with the abstract syntax of Henshin transformation rules because the abstract syntax is used to create new transformation rules on the fly. Contrasingly, our approach makes use of the *concrete syntax* of ATL.

The authors of (Tisi et al., 2010) present several proposals to facilitate the definition of HOTs in ATL. In a survey (Tisi et al., 2009) the authors detected four usage classes of HOTs in real-world model transformation scenarios. Like in the EMF Henshin approach, the authors use ATL to dynamically create new ATL transformation rules which implement the higher-order behavior. In contrast to our approach, the abstract syntax of ATL has to be known by the developer of the HOTs. Furthermore, the metamodel has to be fixed at the time when the HOT is written.

In (Oldevik and Haugen, 2007) an approach is presented that also uses HOTs in the field of software product lines. It differs from our work presented in this paper in several ways: First of all, Oldevik and Haugen extend MOFScript, a M2T language, with aspects tailored towards the generation of product specific application code from a product line UML model. We use model-to-text transformations to generate code for a specific model-to-model transformation. The variability that Oldevik and Haugen address in their paper is restricted to variability in the implementation technique. E.g. they use different mechanisms to implement UML associations in the resulting source code. Our approach provides much more flexibility since we can deal with variability in the domain model. In our approach, the variability is bound when meta-code generation is used to derive a configured domain model (which is then used as a basis for generating the application source code). Furthermore, our approach can handle any Ecore-based domain model and is not limited to UML only.

In (Botterweck et al., 2009), the authors describe their approach on using HOTs as a variability mechanism for embedded systems. Botterweck et al. also use HOTs to derive products in a software product line. In their approach, they used it on Matlab/Simulink models. The main differences to our approach are the fact that they (ab)use the M2M transformation tool in order to perform a M2T transformation. Their higher-order ATL transformation only consists of queries, which produces one large output string: the final ATL transformation. Furthermore, they need a second ATL transformation for which the helpers generated by the first one are manually overloaded. Contrastingly, our approach has a clear separation of concerns, as we use a dedicated model-to-text transformation tool, to generate our ATL rules corresponding to the respective domain metamodels. Our approach does not either require any manual overloading of the generated rules or helpers. Furthermore, the approach of Botterweck et al. needs a post-processing step after the transformation has been performed, to remove dangling edges. This is not neccessary in our approach, since the validity of the configured domain model is ensured by sophisticated mechanisms incorporated in our toolchain (Buchmann and Schwägerl, 2012).

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

5 CONCLUSION AND FUTURE WORK

Higher-order transformations have just recently been addressed in model-driven software development (Tisi et al., 2010). As a consequence, there is no sufficient tool support available at the moment. In our paper, we presented a novel approach to bridge this gap by a two-step meta-code generation approach which enables the specification of a higher-order endogenous model transformation that is independent from the concrete metamodel. Only at runtime, ATL code is generated for the specific metamodel using an Acceleo template, and the resulting first-order ATL transformation is then performed on corresponding model instances.

We applied our approach to the problem of product derivation in model-driven software product lines, where we could successfully test its correctness by comparing results to the existing, Java-based solution. The meta-transformation code we had to write is significantly more declarative. Furthermore, the Acceleo template comprises 124 lines of code, which is about a third of the original Java source code (370 lines).

For future work, we plan to transfer our approach to another research project which is dedicated to three-way merging of models (Westfechtel, 2012) and has a problem definition similar to the one described in this paper. The algorithm allows merging of arbitrary Ecore model instances. The main differences are that first, the result is not only obtained by a conditional copy of one, but three input models, and second, filtering of artifacts is replaced by dedicated merge rules.

ACKNOWLEDGEMENTS

The authors want to thank Bernhard Westfechtel for his valuable comments on the draft of this paper.

REFERENCES

Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: Advanced concepts and tools for in-place emf model transformations. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*, LNCS 6394, pages 121–135, Oslo, Norway.

Biermann, E., Ermel, C., and Jurack, S. (2010). Modeling the "Ecore to GenModel" transformation with EMF Henshin. In Mazanek, S., Rensink, A., and Gorp, P. V., editors, *Proc. Transformation Tool Contest 2010 (TTC'10)*.

Botterweck, G., Polzer, A., and Kowalewski, S. (2009). Using higher-order transformations to derive variability mechanism for embedded systems. In Ghosh, S., editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 68–82. Springer.

Buchmann, T. and Schwägerl, F. (2012). Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, pages 37–44, New York, NY, USA. ACM.

Buchmann, T. and Schwägerl, F. (2012). FAMILE: tool support for evolving model-driven product lines. In Störrle, H., Botterweck, G., Bourdells, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., and Tolvanen, J.-P., editors, *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*, CEUR WS, pages 59–62, Building 321, DK-2800 Kongens Lyngby. Technical University of Denmark (DTU).

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Boston, MA.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, 72:31–39. Special Issue on Experimental Software and Toolkits (EST).

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute.

Mens, T. and van Gorp, P. (2006). A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142.

Oldevik, J. and Haugen, Ø. (2007). Higher-order transformations for product lines. In *SPLC*, pages 243–254. IEEE Computer Society.

OMG (2008). *MOF Model to Text Transformation Language, Version 1.0*. OMG, Needham, MA, formal/2008-01 edition.

OMG (2011). *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1*. Object Management Group, Needham, MA, formal/2011-01-01 edition.

Tisi, M., Cabot, J., and Jouault, F. (2010). Improving higher-order transformations support in ATL. In Tratt, L. and Gogolla, M., editors, *ICMT*, volume 6142 of *Lecture Notes in Computer Science*, pages 215–229. Springer.

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. (2009). On the use of higher-order model transformations. In Paige, R. F., Hartman, A., and Rensink, A., editors, *ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer.

Westfechtel, B. (2012). Merging of EMF models - formal foundations. *Software and Systems Modeling*. Online First.