Generating Multi-Variant Java Source Code Using Generic Aspects

Sandra Greiner and Bernhard Westfechtel
Chair of Applied Computer Science I, University of Bayreuth, Universitätsstrasse 30, 95440 Bayreuth, Germany

Keywords: Model-Driven Software Engineering, Model-to-Text, Model Transformations, Software Product Line Engineering, Feature Annotations, Negative Variability, Feature Propagation.

Abstract: *Model-driven product line engineering (MDPLE)* aims at increasing productivity when realizing a family of related products. Relying on *model-driven software engineering (MDSE)* seeks to support this effect by using models raising the level of abstraction. In MDSE model transformations are successfully applied to transform in between different (model) representations. During MDPLE models based on *negative variability* are augmented with variability information, we refer to as *annotations*. To derive products, source code is generated in model-to-text (M2T) transformations. However, applying *single-variant model transformations (SVMT)* to annotated models, typically loses the information in the output as SVMTs are not capable to process annotations. This work contributes a solution which *reuses* existing SVMTs without changing them and which transfers annotations to the output *orthogonally* to the reused transformation. In particular, we contribute generic aspects, supporting any kind of input metamodel, to augment the outcome of (M2T) SVMTs with annotations. Producing *multi-variant source code (MVSC)* is advantageous because all variants are reflected in the output. Thus, changes made inside the MVSC can be integrated easily in all concerned products upon their derivation. Otherwise, this needs to be done manually in a cumbersome process contradicting the purpose of MDPLE, to raise productivity.

1 INTRODUCTION

Model-driven software engineering (MDSE) is a discipline aiming to increase productivity by generating source code from models residing on a higher level of abstraction than the pure text (Völter et al., 2006). Different kinds of models with well-defined syntax and semantics come into play during the process of establishing a software system. A *metamodel* defines the abstract syntax to which a model conforms. The Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) is the de facto standard to realize domain specific metamodels based on the Ecore (meta-metamodel).

Furthermore, a whole set of similar software systems can be summarized in a *software product line (SPL)*. Based on the principles of *organized reuse* and *variability*, the discipline of *SPL engineering (SPLE)*, too, seeks to increase productivity when realizing a family of related products. A well-known process in SPLE (Pohl et al., 2005) is divided into two phases: *domain engineering*, where common parts of the software are defined in a *platform*, and *application engineering*, where products are (automatically) derived from the platform. A platform holding all variants is said to rely on *negative variability* whereas with *positive variability* a base version for all products is extended with different fragments realizing the specifics of single features. Combining both disciplines, MDSE and SPL (*model-driven product line engineering (MDPLE)*) is meant to further raise the level of abstraction and productivity. In MDPLE typically products are derived from models, which are used to design the platform during domain engineering. These *domain models* usually capture all variants without explicitly being aware of the variability. An MDPLE tool based on negative variability is *FAMILE* (Buchmann and Schwägerl, 2012). It uses *feature models* (Kang et al., 1990) to define the variability where a *feature configuration* selecting the features needed in one variant allows to derive customized products from the domain models. A mapping, denoted as *annotation* in the sequel, can be associated with each model element to declare to which feature(s) the element belongs and is necessary for automatically deriving products.

On the other hand, *model transformations* play a key role to automate the creation or evolution of models from models residing on different levels of ab-

Greiner, S. and Westfechtel, B.
Generating Multi-Variant Java Source Code Using Generic Aspects.
DOI: 10.5220/0006536700360047
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 36-47
ISBN: 978-989-758-283-7

stractions. While *model-to-model (M2M)* transformations allow to transform in between two (or more) models, *model-to-text (M2T)* transformations generate plain text (output) from a given model (input). M2T transformations are often used to automatically generate source code for a given model. A transformation considering only one direction is called *unidirectional* whereas languages which not only specify the *forward* direction but also treat the *backward* transformation are called *bidirectional*. While in *inplace* transformations the input model serves as output model as well, a separate output is created (*batch*) or modified (*incremental*) in *out-place* transformations. Incremental transformations usually rely on keeping track of the link between in- and output element in *traces*.

A frequently used M2M language is the *Atlas Transformation Language (ATL)* (Jouault et al., 2008). *Acceleo*[1], is a M2T language implementing the Mof2Text standard proposed by the Object Management Group (OMG) (Object Management Group, 2008). Likewise, *Xpand* (Efftinge et al., 2004) is another M2T language originating from the open Architecture Ware (oAW) project with the special feature to support aspect-oriented programming (Kiczales et al., 1997) which allows to extend already existing transformations without modifying the original one.

By now, model transformations in various forms are mature and frequently applied in MDSE. In MDPLE, however, multi-variant models are associated with annotations which are not covered by aforementioned *single-variant model transformation (SVMT)*. Having a multi-variant input insofar as all variants are visible in the model, the outcome also includes all variants but the annotations are lost. If the multi-variant output should be annotated as well, usually the annotations have to be manually transferred in a laborious and error-prone process.

Recently, the problem of this kind of *multi-variant model transformations (MVMT)* was addressed in research: In (Salay et al., 2014) the authors change the execution semantics of single-variant graph-based M2M transformations to conform to annotated multi-variant input and produce annotated multi-variant output. Contrastingly, an *a posteriori* approach to automatically apply annotations after executing the SVMT is proposed in (Greiner et al., 2017). Still, this work only supports the *reuse* of specific ATL transformations. Other research rather regards variability in the rules (Strüber and Schulz, 2016) sometimes resulting in the need to learn new language constructs (Sijtema, 2010).

The enumerated proposals, however, only cover M2M transformations: source code is usually only derived for single products by providing a set of features. If manual changes, e.g., implementing a method body, are performed in the product, these modifications need to be applied to every single product separately. This in turn is laborious and might lead to diverging or erroneous realizations of the same feature.

Therefore, the present work contributes a solution which successfully (1) *reuses* SVMTs and augments the outcome (2) *orthogonally* to the existing transformation with annotations. Technically, the language Xpand[2] is chosen since it leaves existing SVMTs untouched and allows to alter them with aspects. In contrast to aspect-oriented approaches, e.g., written in Xpand (Voelter and Groher, 2007), we do not implement one aspect for each feature which are added on demand to the final product (*positive variability*). Rather annotations are propagated to the source code, reflecting all variants, by using one single generic Xpand aspect. The annotations are integrated in form of preprocessor comments resulting in annotated *multi-variant source code (MVSC)*. Consequently, manual changes can be added to the MVSC and are included in the concerned products when being derived by providing a set of features to the preprocessor.

The remainder is structured in the following way: At first, we motivate the need for MVSC based on a concrete example. Then, the paper provides an overview on the contribution and, next, a description of the technical realization. Finally, it shows example transformations. In the end, related work is discussed and a conclusion is drawn including an outlook on future work.

2 MOTIVATION

This section stresses the importance of MVMT, in general, and of annotations inside source code, in particular. By looking at a simplified form of the well-known SPL for *Graph* libraries as example from literature (Lopez-Herrejon and Batory, 2001), we deduce some properties and requirements an (M2T) MVMT should fulfill.

[1] http://www.eclipse.org/acceleo

[2] We used the Xpand2 implementation which we refer to as *Xpand* in the sequel.

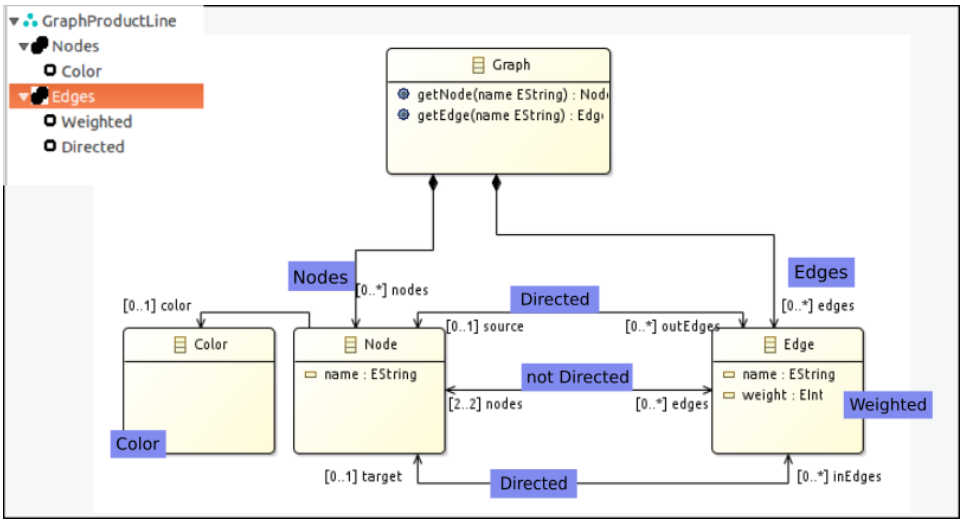

Figure 1: Graph domain model annotated with features from the feature model seen in the upper left corner.

2.1 Necessity for Annotated Multi-Variant Source Code

The example is based on an annotated Ecore model for the Graph SPL. As depicted in Figure 1, the Graph consists of Egdes and Nodes which are mandatory features of the feature model placed in the upper left corner of the figure. Edges might be Directed and/or Weighted. Moreover, nodes might be Colored.

With state-of-the art MDPLE tools based on negative variability, so far, products are derived by providing a feature configuration, first, and by filtering the annotated multi-variant domain model afterwards. Based on the filtered model, (Java) source code is generated resulting in *single-variant source code (SVSC)*. Having, for instance, only the features Edges and Nodes selected results in three classes. The class Node contains a field edges and a corresponding field nodes is included in the class Edge. For both fields respective accessor methods are generated. Since for the omnipresent Graph class only the method stubs for the operations getNode() and getEdge() are created, the engineer implements their method bodies *manually* in the final product. The whole code, however, is completely *unaware* of features and the manual modifications are unique to this single product.

In the meantime, more products are derived, all missing the two method bodies implemented in the first derived product. As the manual implementation should be present in all products, the developer copy-and-pastes the manual modifications to all other derived products. Despite the increased effort – contradicting the promise of increased productivity when using MDPLE –, a high risk to forget one product or to slightly modify the implementations inadvertently remains.

This problem could be solved by generating the MVSC first and adding manual implementations there. Although it is possible to generate the source code including all variants from elements present in the domain model, the default Ecore code generation is not able to integrate annotations in the generated text. A link to the feature(s) a single source code fragment is realizing is missing. As a consequence, it is not possible to derive products from this code base automatically. To do so, it would be necessary to have the annotation belonging to a source code fragment associated with the corresponding fragment. Please note: This kind of problem can hardly be solved by designing more detailed domain or feature models. Even if behavior was modeled and source code for method bodies was automatically derived, still typically manual extensions are required. If such extension was associated with a feature, it would be best having it in all corresponding products automatically upon derivation.

As deducted from this scenario, having annotated MVSC, hence, provides the following benefits:

- Products can be derived automatically from the MVSC without generating a single-variant model first.
- Derived products may include manually added source code fragments concerning them, *increasing the degree of automation and productivity*.

2.2 Consequences

In order to be beneficial, we postulate some requirements on (text producing) MVMTs:

Reuse. Existing SVMT should not be meaningless but integrated and ideally be *reused* as they are. This is achieved by propagating annotations separately and automatically. This approach offers the following advantages: First, the programmer does not have to learn any new language construct. Second, the cognitive level of the transformation does not increase. Third, already achieved realizations are not meaningless but are integrated seamlessly.

Tracelink. The kind of annotations realized in the input must be translated into a form understood by the output which should happen *orthogonally* to the reused SVMT. Having the links transferred in an orthogonal step, strictly separates the concerns of transforming the model and propagating the annotations.

Commutativity. The source code generated for a model which was derived from a multi-variant domain model by specifying a feature configuration should be the same as the source code which results from preprocessing the annotated MVSC by the same configuration. Deriving the source code from the MVSC achieves the same goal but based on a higher degree of automation (see Section 2.1).

Transformation Engine. Furthermore, an MVMT engine is needed that fulfills the following criteria:

- **Generic:** support any kind of variability representation as input and produce generic output based on the reused transformation
- **Reuse of:** already present SVMTs.

On the whole, we postulate MVMTs which are automatically executed based on the reuse of unchanged SVMTs. Supporting commutativity is achieved by propagating tracelinks orthogonally to the reused SVMTs.

3 OVERVIEW

This section provides an overview on our solution towards generating MVSC: SVMTs should be executed as they are and augmented with annotations *orthogonally* to the SVMT. In this work the source code produced by an M2T transformation should be annotated without changing the SVMT allowing to derive products from the multi-variant code base.

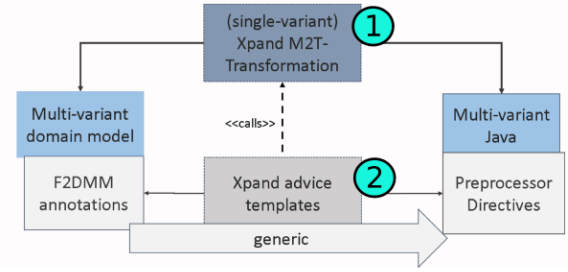

Figure 2: Realization approach to an *orthogonal* variability integration.

Figure 2 depicts our approach schematically: An SVMT should be *reused* (step 1) and annotations should be translated separately (step 2) into preprocessor directives. Since the SVMT should not have to be altered, the approach demands for an M2T transformation language that generates artifacts or provides mechanisms to work with orthogonally to the executed base transformation. We have picked the M2T language Xpand to serve this purpose because it allows to adapt the behavior of existing model transformations with an aspect-oriented approach. More details on the language are given in the remainder of this section.

As shown in the aforementioned figure, the annotation transfer relies on a generic implementation supporting arbitrary model transformations written in Xpand: F2DMM annotations (explained in Section 3.3) are transferred to the resulting source code in form of preprocessor directives. As Java does not support a preprocessor, the directives are translated into JavaDoc comments and will be processed by a separate Eclipse plugin, which is roughly explained in Section 3.2. Thus, source code fragments belonging to a specific feature are embraced with comments in the multi-variant Java source code resulting from the SVMT.

Please note: The reused SVMT is single-variant as it does not cover any variability information at all, being *unaware* of the variability. Nevertheless, the input domain model is multi-variant as it may reflect the complete platform including all variants, based on the principle of negative variability. However, the domain model elements are likewise unaware of the variants they correspond with. This information is captured orthogonally to the domain model in a separate model, e.g., in form of F2DMM annotations (see Section 3.3). Nevertheless, generating source code from the multi-variant domain model in the SVMT results in MVSC including all modeled variants of the input. The MVSC is unaware of the corresponding

features, too.

Altogether, with the provided solution it is possible to support the automatic MVSC generation for a single-variant Xpand code generation written for an Ecore-compliant metamodel. With the present solution the input models must store the annotations in the F2DMM. However, for a different annotation representation a new advice could be written to support other SPL tools.

3.1 Xpand

Xpand stems from the openArchitectureWare (oAW) project where it serves as template-based M2T transformation language. Extensive documentation can be found in (Efftinge et al., 2004).

Xpand projects typically consist of *templates*, *extensions* and an MWE(2) *workflow*. In the templates a DEFINE block specifies how a metamodel element is converted into text. In these blocks plain text can be intermingled with other Xpand directives, e.g., the FILE or EXPAND directives, where the first one creates a file with the given path and the latter one calls another DEFINE block. Moreover, complex instructions can be written in extensions which provide a subset of the Xtend language. Additionally, extensions may call Java methods that could be used for any instruction which is not (or only restrictively) expressible with Xtend.

A basic example of a text producing template is shown in Listing 3.1: For all eClassifiers (l. 2) contained in an EPackage (l. 1) a file is created with the path name returned from an extension call (l. 5). The text in line 6 is written verbatim to the output except for the extension call inside the French quotation marks. The call returns a String containing the fully qualified name of the respective package.

```
1 «DEFINE main FOR ecore::EPackage»
2   «EXPAND cl FOREACH eClassifiers»
3 «ENDDEFINE»
4 «DEFINE cl FOR ecore::EClassifier»
5   «FILE packagePath() + "/" + name +
        ".java"»
6   package «packageName()»;
7   «ENDFILE»
8 «ENDDEFINE»
```

Listing 1: Example transformation generating files for eClassifiers.

Most importantly for the present work, the language allows to extend or rewrite template definitions with an aspect-oriented component: *advice* templates define AROUND blocks specifying a generated element that they might (almost) arbitrarily alter. Quite like a DEFINE statement, they first state a fully qualified element that should be modified and second the type of this element. *Wildcards* (*) inside the fully qualified name allow to scan different directories and/or to match arbitrary element types of DEFINE blocks.

Finally, MWE(2) workflows[3] are used to execute the transformation. Workflows are given a name and consist of different components. They are interpreted as normal Java classes. When executing model transformations, a workflow usually specifies a reader component which receives the path to the input model/s and its/their metamodel/s and an Xpand generator component. The latter expects the entry point at the transformation (corresponding with the root element of the input) and a path to which the output is written. The generator component may receive advices which can be realized for templates and extensions alike.

In addition, Xpand allows to realize incremental behavior with protected blocks in the templates and by specifying the respective files in the workflow. However, this functionality was not examined in our present contribution.

3.2 Java Preprocessor

In order to properly handle source code annotated with preprocessor directives, some facility supporting this task is needed. In the past, some approaches have been proposed but they are mostly not compatible with current Eclipse projects, e.g., *Prebop*[4], or require a manual adaptation of the build process, e.g., when using the *Java Comment Preprocessor*[5]. Thus, we introduce a different preprocessor, which we developed in a model-driven way, realized as Eclipse plugin. As Java does not support preprocessor directives, they are included in a non-invasive way into the source code in the form of comments. The preprocessor is *configurable* with respect to

- the files that should be preprocessed by their file-ending,
- the kind of comment signs that open and close the directives inside these files,
- the names/signs for the opening, closing and possible branching directives.

Furthermore, the preprocessor expects a set of flags with boolean values which resemble a feature configuration. All source code fragments surrounded with

[3] http://www.eclipse.org/Xtext/documentation/306_mwe2.html

[4] http://prebop.sourceforge.net/

[5] https://github.com/raydac/java-comment-preprocessor

directives evaluating to false are turned into comments by the preprocessor.

3.3 Feature to Domain Mapping Model

Last but not least, a representation for the variability of models is needed. We choose the form provided by the tool FAMILE (Buchmann and Schwägerl, 2012), which is based on negative variability and relies on the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009). While feature models capture the variability, a single product is associated with one feature configuration which sets the features to true or false. Moreover, annotations are boolean expressions over the features, called *feature expressions(FE)*, and can be assigned to arbitrary domain model elements. They are stored in a separate model, the *feature to domain mapping model (F2DMM)* which resembles the structure of the domain model it belongs to and, thus, stores an *object mapping (OM)* for all EObjects contained in the domain-model. For example, every instance of an EClass or an EOperation, etc., is referenced by an OM and may be visible just for specific features. Based on feature configurations, final products are derived by filtering all objects the FEs of which evaluate to false.

4 REALIZING THE MULTI-VARIANT SOURCE CODE GENERATION

As mentioned before, this work aims at providing a mechanism to augment the result of a single-variant M2T transformation – in the present approach written in Xpand – with annotations. Particularly, we propagate FEs stored in the separate mapping model of the tool FAMILE (F2DMM) resulting in annotated MVSC. We assume the Xpand SVMTs *already exist*. They should be reused without any modifications.

For the solution two components are decisive: A template containing the *advice* to modify the existing SVMT and a *second MWE(2) workflow* executing the workflow of the SVMT augmented with the provided advice. Now details on both are illuminated.

4.1 Advice Template

First of all, the automatic transfer of annotations requires a generic instruction to annotate source code fragments related with an annotated model element. The generated text should be surrounded with a comment opening the IF preprocessor directive stating the respective annotation and a second comment closing the preprocessor directive after the generated text is written to the output.

First, in order to regard any model element text is generated for, the provided AROUND statement expects an arbitrary Object. Next, it searches the respective object mapping in the F2DMM referencing the domain model in which the given EObject is contained. If the OM stores an FE, an opening and a closing comment need to encompass the actual text production of the SVMT.

Listing 4.1 depicts a simplification of the respective advice: By using wildcards in the AROUND statement (l. 1), the following procedure is executed for any Object text is generated for inside a file located in a *templates*[6] folder. As FAMILE only supports Ecore-compliant domain models, EObjects are the only elements, text is generated for. Thus, firstly, the algorithm searches the corresponding OM (l. 2) with the help of an extension call. The respective function getOM examines the F2DMM, which is loaded in the background upon the first call of getOM. It, then, returns the matching mapping. If the OM contains an FE (l. 3), the actual text production is surrounded by comments containing the FE (l. 4 - 7). The comment is constructed in another function provided in the extensions (l. 4 and 7) and depends on the fact whether the container of the object also owns an FE. The instruction targetDef.proceed() (l. 6 and 10) executes the original text production of the SVMT.

```
1  «AROUND templates::*::* FOR Object»
2    «LET getOM((EObject)this) AS om»
3      «IF om.isSetFeatureExprStr()»
4        «getOpeningComment()»
5        // @ID: «getID()»
6        «targetDef.proceed()»
7        «getClosingComment()»
8      «ELSE»
9        // @ID: «getID()»
10       «targetDef.proceed()»
11     «ENDIF»
12   «ENDLET»
13 «ENDAROUND»
```

Listing 2: Simplification of the advice encompassing an actual text genreation.

By adding the comments, it is possible to relate the text surrounded with comments to a specific feature and, hence, to inject the variability information in the pure text. Note, that each element receives only the FE it is associated with in the F2DMM. As there

[6]Per convention, Xpand projects store all the text generating templates in a folder *templates* and extensions in a folder *extensions*.

are no nested preprocessor directives by construction, transitive dependencies are not reflected in the created comment style. Up to now, we assume a valid feature model and, moreover, prefer to put only the concrete FE. In that way, the information remains which concrete feature a single element realizes.

Furthermore, in any case a unique ID is generated for the EObject (l. 5 and 9) and put before the generated text. This way we keep a unique trace information, which might be used to support later evolution processes.

Listing 4.1 illustrates the comment style, for instance for the field weight of the class `Edge` from the example shown in Section 2. First, the container expression (Edges) is closed (l. 1) and the actual expression Weighted is opened (l. 2) in the opening comment. In the closing comment the actual preprocessor directive is closed (l. 6) and the one of the container is opened again (l. 7).

Closing and opening container expressions is necessary as nested directives are not supported with the present approach: One closing JavaDoc comment always closes all comments that were opened before. Hence, closing the comment originally associated with the EObject would also close the one of any container, if it was still open. Therefore, if the container owns an FE, its statement must be opened after closing the one associated with the EObject. Additionally, it must be closed before opening the actual comment because the preprocessor does not support nested comments.

```
1 /** #ENDIF Edges */
2 /** #IFDEF Weighted # */
3 // @ID: _TACGEFc3Eeetfr4BhVYAxQ
4 protected int weight;
5
6 /** #ENDIF Weighted*/
7 /** #IFDEF Edges # */
```

Listing 3: Example of a field annotated with JavaDoc comments generated by our advice.

4.2 Workflow

The second necessary part to realize the annotations is the workflows. As depicted in Figure 2, the Xpand project providing the advice template calls the reused SVMT. Thereby it adds the generic advice template to the Xpand generator executed in the single-variant workflow. Accordingly, the advice project provides a workflow that executes the SVMT and adds the previously presented advice. Without the loss of generality[7], technically, we assume that the single-variant code generation is initiated by an MWE2 workflow providing a slot to add an advice template. This slot is filled with the generic Xpand advice project which, finally, executes the single-variant workflow augmented with advices.

Summing it up, in order to execute the MVSC generations, the source domain model must be provided as input to the reused MWE2 workflow which is executing the single-variant code generation. Alongside for any EObject, text is generated for, the advice is being called surrounding the text with the corresponding comment.

5 EXAMPLES

To demonstrate the feasibility of the contributed advice, we present two independent transformations respectively. The first one produces Java source code for annotated Ecore models, the second one produces Java source code for annotated instances of the Java MoDisco metamodel. MoDisco (Bruneliere et al., 2010) is an extensible model-driven framework supporting different use cases of software modernization. It provides an Ecore-compliant metamodel that resembles the Java AST and offers discoverers that, e.g., translate Java source code into valid instances of this metamodel. The framework also includes an M2T engine that converts such models back into Java source code.

The transformations were partially reimplemented in Xpand for the purpose of this evaluation because the EMF and MoDisco code generators are implemented in different template languages (JET and Acceleo, respectively, both do not natively support aspect-orientation). In addition, since we do not examine the incremental behavior of Xpand yet, the M2T templates focus on the structural parts of the metamodels and neglect, e.g., method bodies.

Below, we present how the outcome of the aforementioned transformations is enriched with annotations by our provided generic aspect. We pick one Ecore class as example and illustrate the transformation process: Remember the class Edge, depicted in Figure 3 on the left hand side which is visible when the feature Edges is selected and which may have an attribute weight (Weighted) and a source and target node when feature Directed is selected or exactly two nodes if it is not. We only show the generated interface for space and simplicity reasons.

[7]The procedure is quite similar for a simple MWE workflow.

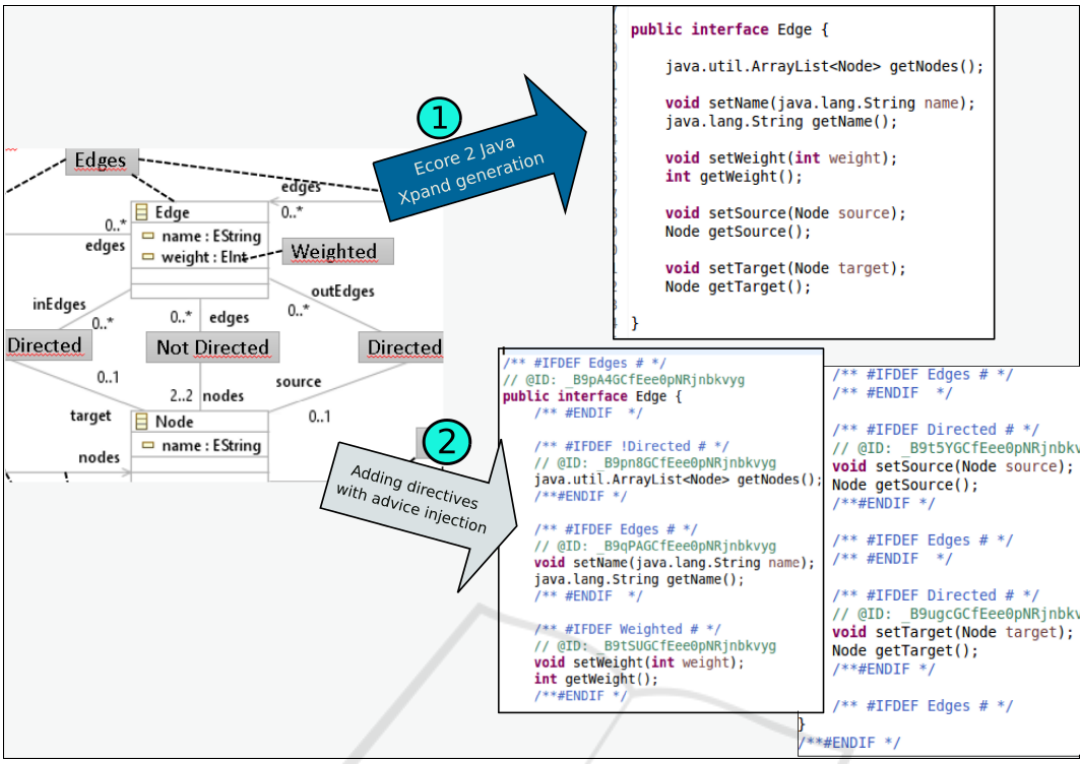

Figure 3: Generated interface when applying the Xpand transformation without (1) and with (2) advices on the Edge class.

Ecore to Java Source Code. The first transformation receives as input an annotated Ecore model of the Graph SPL (Lopez-Herrejon and Batory, 2001). The plain Xpand transformation creates source code which partially resembles the default Ecore code generation: For instance, for every EClass an interface and a class are created. Additionally, for the structural features fields are included in the containing classes and set- and get-method stubs in both, the interfaces and classes.

Figure 3 shows the different outcome when applying the Xpand transformation without (1) and with (2) advices. Applying the SVMT (1) on the multi-variant edge class results in a class and an interface where all modeled elements – independent of the associated features – are visible. The connection to the features is lost.

Executing the same transformation *with advices* (2), i.e., our provided second workflow, generates the same source code but adds the annotations in form of preprocessor directives. For instance, the methods setWeight and getWeight are embraced with the "# IFDEF Weighted #" comment and the closing "# ENDIF" comment.

In general, each element created in a DEFINE block receives the ID corresponding with the EObject it was created from. Moreover, if its associated OM contains an FE, it is embraced with a preprocessor directive mentioning the FE. Accordingly, while the attribute name inside the edge class only receives its corresponding ID, the other methods are additionally embraced by preprocessor directives. Consequently, the resulting source code is aware of all possible variants and allows to associate manual modifications with a feature.

Finally, products can be derived from the MVSC. Figure 4 depicts the edge interface after the preprocessor was run on a configuration that consists of nodes with weighted edges: All code with deselected features is moved into JavaDoc comments, hence, making it invisible in the resulting product[8].

MoDisco to Java Source Code. In order to illustrate the same advice is able to annotate source code generated for instances of different metamodels, we additionally investigated an annotated Java MoDisco model of the Graph SPL. The model discovered from the source code resulting from the default Ecore code generation serves as input to the second scenario. It

[8]Still, the complete MVSC is part of a product to allow working in continuing steps with different configurations and possible modifications. To export only the visible source code into a new project as standalone product would not cause a problem.

Figure 4: The Edge interface after running the preprocessor on the feature configuration.

was annotated with an F2DMM containing similar presence conditions as could be inferred from the annotated Ecore elements. A snippet of the annotated Edge interface in the F2DMM tree perspective is displayed in Figure 5 respectively for the other elements. Annotations associated with one element are written verbatim behind the respective element in grey color. In contrast to the Ecore model, in the Java model resulting from discovering the source code, for example, the accessor methods created for a field are separate model elements. Accordingly, they are associated with distinct OMs, that, however, carry the same FEs in the F2DMM.

On the left of Figure 6, the annotated source code resulting from executing the SVMT with our provi-

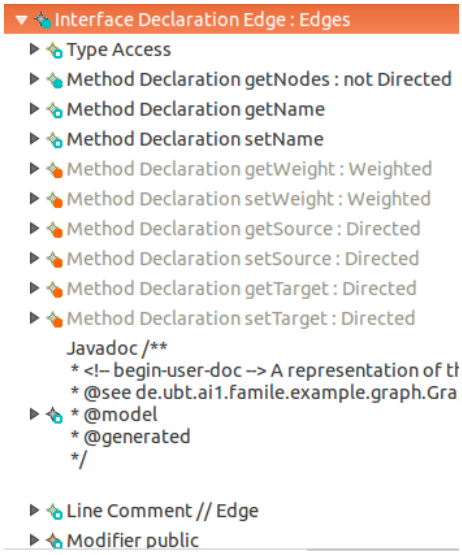

Figure 5: Annotated MoDisco Edge interface.

Figure 6: The annotated Edge interface after the MVMT (left) and after running the preprocessor (right).

ded Xpand advice is placed. The source code almost equals the one generated with the Xpand Ecore code generation (bottom of Figure 3). However, the Xpand implementation generating Java from Ecore in the first scenario does not include all details of the Ecore standard code generation. Rather it uses the standard Java libraries (e.g., using an ArrayList instead of the EList). As a consequence, the discovered model resulting from the default Ecore code generation (which is input to this scenario) includes EMF-specific types which are, thus, included in the text produced with the MoDisco to Java Xpand transformation.

Comparing the Outcome. As the examples above demonstrate, one and the same advice presented in Section 4 is sufficient to correctly translate FEs into preprocessor directives and to place them around the corresponding generated text. Thus, the methods created for the field weight in the interface Edge are given the corresponding comment. Similarly, the met-

hods for source and target are only present in Directed graphs. Some small differences are visible. For instance, with the MoDisco model as input, each accessor method receives a separate comment instead of commenting them pairwise. In addition, their parameters are given own comments. This is due to the fact, that the MoDisco model contains a MethodDeclaration for each operation and handles the parameters as unique EObjects. As they are all considered as EObjects, the F2DMM foresees a mapping for each of these elements and, accordingly, they may contain separate FEs that are added with the contributed advice. In contrast, in Ecore the corresponding mapped object is a structural feature (one EObject) for which two methods with a fixed style are created in *one* DEFINE block.

Finally, we like to stress that applying the same feature configuration results in (almost) the same product. Some differences remain, like having an EList instead of an ArrayList as container for multi-valued structural features. Nonetheless, the same functionality is achieved.

Conclusion. The examples showed that it is possible to automatically annotate source code with our multi-variant M2T Xpand transformation by reusing SVMTs. Our contribution aims at providing a generic mechanism to propagate annotations into source code, which was demonstrated for two transformations based on considerably different metamodels. The success of transferring the annotations, however, depends on whether the text for some EObject is created in a separate DEFINE statement. If this is the case, we can deduce that all metamodels annotated with FEs could be supported. Accordingly, the reused SVMT must adhere to some (standard) Xpand coding conventions. On the whole, our contributed generic advice allows to generate multi-variant source code from which different products can be derived.

6 RELATED WORK

The present work realizes an M2T MVMT automatically propagating annotations from models based on negative variability to source code. Regarding such transformations, the field of comparable work is populated rather sparsely.

In (Greiner et al., 2017) the authors *reuse* existing single-variant M2M transformations and transfer annotations automatically, based on evaluating a persisted trace and the execution model, in a second step orthogonally to the reused model transformation. We take up the idea of reusing existing model transformations but add annotations with the help of aspects. Thus, we do *not* have to evaluate the transformation artifacts *a posteriori* and do not rely on a persisted trace. Rather we integrate annotations based on the capabilities of the transformation language and provide a generic implementation that, however, also allows to reuse existing transformations without modifying them.

Another approach to realize MVMTs *lifts* the transformation (Salay et al., 2014). This work proposes an algorithm for graph-based transformations which generates an MVMT where the single-variant rules, e.g., their application conditions, are interpreted with variability semantics. The approach was not only successfully applied to in-place graph transformations but also to out-place M2M transformations with a graph-oriented DSL that had to be slightly modified for this purpose (Famelis et al., 2015). Instead of changing the execution semantics of the SVMT, our approach executes the single-variant transformation without adaptations. We rather exploit the existing capabilities of the engine without changing the SVMT to propagate annotations seamlessly. Furthermore, we produce source code instead of models as it increases the degree of automation to deliver a final product. Manual modifications required in the annotated MVSC may be automatically integrated in all products in that way.

In contrast, in (Strüber and Schulz, 2016) the transformation rules themselves are variability-aware. Although the proposed tool allows filtered editing on the rules which reduces the cognitive complexity, it is necessary to consider variability in the transformation rules themselves. Moreover, the tool solves different problems, e.g., allowing to transform in between platform independent and platform specific models in the context of *Model-Driven Architecture (MDA)* (Mellor et al., 2004).

In (Sijtema, 2010) the authors are, likewise, changing the transformation rules by introducing new syntax to ATL transformations. The new language constructs are maintained in *higher-order transformations*. Again, this approach addresses variability in transformations rather than in models.

Technically, Xpand has already been used to enrich existing transformations with source code fragments belonging to different features (Voelter and Groher, 2007). In contrast to our contribution, a different approach is applied. As with aspect-oriented approaches, each feature that could be added to a common code base is realized in a separate aspect and provided in a second workflow. We, instead, provide only one single aspect that annotates source code contai-

ning realizations for all features. Thus, our approach is tailored for negative rather than for positive variability.

On the whole, our approach is unique with respect to the following aspects: First, it addresses M2T transformations rather than M2M transformations. Second, unlike in (Strüber and Schulz, 2016; Sijtema, 2010) the approach deals with multi-variant input rather than with multi-variant transformation specifications. Finally, unlike in (Salay et al., 2014; Greiner et al., 2017), the already existing functionality of the transformation engine is exploited, relying on aspects which extend reused transformation definitions in a modular way.

7 CONCLUSION AND FUTURE WORK

Summing it up, the presented approach provides an automated propagation of variability annotations by reusing SVMTs. In particular, we transfer FEs to the MVSC generated from Xpand templates. To the best of our knowledge, it is the first approach supporting the automated creation of annotated multi-variant source code by reusing (existing) single-variant M2T transformations. Thus, with the present work final products can be derived from multi-variant source code instead of from single-variant models. This, in turn, lays the foundation to automatically integrate manual modifications to the source code in all derived products. Previously, this had to be accomplished with laborious handwork on every single product which leads to errors and an increased effort that contradicts the paradigm of MDPLE of increased productivity.

Future work investigates the incremental behavior of Xpand transformations when altering it with advices and considers to propagate manual changes in a backward transformation to the corresponding model elements. Moreover, the combination of FEs, especially when considering backward transformations, should be examined.

REFERENCES

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering (ASE 2010)*, pages 173–174, Antwerp, Belgium.

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

Efftinge, S., Friese, P., Hase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., et al. (2004). Xpand documentation. Technical report, Technical report, 2004-2010.(cited on page 64).

Famelis, M., Lúcio, L., Selim, G., Di Sandro, A., Salay, R., Chechik, M., Cordy, J. R., Dingel, J., Vangheluwe, H., and Ramesh, S. (2015). Migrating automotive product lines: a case study. In *International Conference on Theory and Practice of Model Transformations*, pages 82–97. Springer.

Greiner, S., Schwägerl, F., and Westfechtel, B. (2017). Realizing multi-variant model transformations on top of reused ATL specifications. In Pires, L. F., Hammoudi, S., and Selic, B., editors, *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 362–373, Porto, Portugal. SCITEPRESS Science and Technology Publications, Portugal.

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.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. *ECOOP'97–Object-oriented programming*, pages 220–242.

Lopez-Herrejon, R. E. and Batory, D. S. (2001). A standard problem for evaluating product-line methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, pages 10–24, London, UK. Springer.

Mellor, S. J., Kendall, S., Uhl, A., and Weise, D. (2004). *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

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

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Germany.

Salay, R., Famelis, M., Rubin, J., Sandro, A. D., and Chechik, M. (2014). Lifting model transformations to product lines. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 117–128.

Sijtema, M. (2010). Introducing variability rules in atl for managing variability in mde-based product lines. *Proc. of MtATL*, 10:39–49.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Boston, MA, 2nd edition.

Strüber, D. and Schulz, S. (2016). *A Tool Environment for Managing Families of Model Transformation Rules*, pages 89–101. Springer International Publishing, Cham.

Voelter, M. and Groher, I. (2007). Handling variability in model transformations and generators. In *7th OOPSLA Workshop on Domain-Specific Modeling*.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.