Generic Framework for Evaluating Commutativity of Multi-Variant Model Transformations

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 Transformations, Software Product Lines, Multi-Variant Model Transformations, Annotative Approaches, Evaluating Commutativity.

Abstract: Multi-variant model transformations (MVMTs) aim at automatically propagating variability annotations present in software product lines (SPL) when executing state-of-the-art model transformations. Variability annotations are boolean expressions used in annotative SPL engineering (SPLE) for expressing in which products model elements are visible. Developing the SPL in a model-driven way requires various model representations, e.g., database schemata for data storage or Java models for the code generation. Although model transformations are the key essence of model-driven software engineering (MDSE) and can be used to generate these representations from already existing (model) artifacts, they suffer from not being able to handle the variability annotations. Thus, the developer is forced to annotate target models manually contradicting the goal of both disciplines, MDSE and SPLE, to increase productivity. Recently, approaches have been proposed to solve the problem using, e.g., traces, to propagate annotations without changing the transformation itself. For evaluating the outcome all of the approaches require the transformation to commute w.r.t. the derived products. Although the criterion is the same, a common framework for testing it does not exist. Therefore, we contribute a generic framework allowing to evaluate whether the target model of arbitrary (reuse-based) MVMTs was correctly annotated according to the shared commutativity criterion.

1 INTRODUCTION

Only recently, multi-variant model transformations (MVMTs) have gained popularity combining two disciplines: *software product line engineering* (SPLE) and model transformations, required in *model-driven software engineering* (MDSE) .

SPLE is centered around the paradigms of *organized reuse* and *variability* for increasing productivity when developing a set of closely related products. A common development process is composed of two phases: In *domain engineering* the platform comprising the superimposition of the products is developed. In contrast, in *application engineering* products are derived from the platform and prepared for delivery (Pohl et al., 2005). Feature models (Kang et al., 1990) are a common means to express the discriminating factors of the software as *features*. In annotative approaches (Apel et al., 2009) a superimposition of the products is developed and the artifacts are associated with variability annotations which are boolean expressions over the features. Below we refer to these expressions as *annotations*. By (de-)selecting features in a *feature configuration* final products can be filtered from the superimposed platform where all elements the annotation of which cannot be satisfied under the given feature configuration are removed from the superimposed model.

Model transformations, on the other hand, are the heart of MDSE (Stahl et al., 2006). The various realizations allow for transforming a source (input) model to a target (output) representation. Transformations are categorized with respect to the supported transformation modes: *model-to-model* (M2M) or *model-to-text* (M2T) transformations, creating either a new model representation or text (e.g., source code from templates), *in-place* or *out-place* transformations (having the source model or a different model as target, respectively) as well as *batch* or *incremental* and *unidirectional* or *bidirectional* transformations. During the transformation execution many tools persist the information of corresponding source and target elements in so-called *traces*.

In model-driven SPLE (Czarnecki et al., 2005; Buchmann and Schwägerl, 2012; Heidenreich et al., 2008) models are the main development artifacts. Ap-

Greiner, S. and Westfechtel, B.
Generic Framework for Evaluating Commutativity of Multi-Variant Model Transformations.
DOI: 10.5220/0007585701550166
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 155-166
ISBN: 978-989-758-358-2

plying an annotative approach in model-driven SPLE means that model elements are associated with annotations. In order to automatically convert one type of model of the product line into another representation, typically model transformations are used. For instance, when developing the SPL a UML class diagram (OMG, 2017) may capture the structure of the platform. From the UML representation source code needs to be created. Despite the fact that model transformations allow to perform this task, they are unaware of the annotations attached to the model elements. Consequently, a state-of-the-art model transformation can create the target model but neglects the annotations. Annotating the target model manually is a laborious and error-prone task contradicting the general purpose of MDSE and SPLE to increase productivity. Thus, the task calls for transformations allowing to automatically propagate annotations from the source to the target representation. Solutions supporting this task are referred to as *multi-variant model transformations*.

Recently, different automations to solve the problem of propagating annotations have been published. Most of them try to (re)use already existing (single-variant) transformations and to propagate the annotations orthogonally. The solutions are said to be correct whenever the same products can be derived from the annotated target model as would have been created when deriving the products from the source model and transforming them afterwards. This required property is called *commutativity*. While some approaches are formally proven to fulfill this correctness criterion (Westfechtel and Greiner, 2018; Taentzer et al., 2018; Strüber et al., 2018), others verify the criterion in their specific use cases (Salay et al., 2014; Greiner and Westfechtel, 2018; Greiner et al., 2017). Although all of the approaches have the same correctness criterion in common, there is no general framework to evaluate commutativity, yet. For this reason, in this paper we contribute a generic evaluation framework allowing, on the one hand, for executing reuse-based approaches and, on the other hand, to evaluate whether the annotations of the target model are valid with respect to the filtered products. The framework generalizes the way of how (domain) models and feature models are represented and how the models are filtered. Furthermore, since models can be represented differently, the comparison mechanism is also generalized for arbitrary model types. In this way the framework is able, e.g., to evaluate a trace-based MVMT (creating models as targets) or, quite differently, approaches producing text instead of models as output. Moreover, product lines created by different tools should be testable. Last but not least, the framework not only provides the answer whether commutativity is achieved but points out where violations have taken place. Consequently, it allows to detect misbehavior and to correct wrong annotations or to improve the transformation approach.

In the following section we provide background information on multi-variant model transformation and motivate why a generic framework for their evaluation is required. In Section 3 the architecture of the framework is described and one realization is presented in the following example. Finally, related work is shortly discussed and a conclusion is drawn.

2 BACKGROUND AND MOTIVATION

The importance and need for MVMT has already been motivated in various publications, e.g., (Salay et al., 2014; Schwägerl et al., 2016; Greiner and Westfechtel, 2018; Westfechtel and Greiner, 2018; Taentzer et al., 2018). This section provides details and categorizes the existing approaches. Based on the commonalities of the approaches we determine requirements for our framework to support the evaluation of reuse-based approaches. Furthermore, the section presents correctness criteria to evaluate the outcome of the executed MVMT.

2.1 Existing Approaches

Different approaches for automating the annotation of model elements exist, which can be roughly categorized in *black-box* and *white-box* solutions depending on the fact whether *no* or *all* internals (i.e., the contents of the transformation) are exploited, respectively. All of them, however, have in common that existing transformations or the existing tool environments are reused to some extent.

Lifting (Salay et al., 2014) is one approach changing the semantics of the execution engine and is defined for graph transformations but was also applied in out-place transformations with a graph-like DSL (Famelis et al., 2015). However, lifting requires to know the contents of the transformation specification and to enumerate all rules. Likewise, another solution based on higher order transformations in ATL transformations (Sijtema, 2010) is specific to the language ATL (Jouault et al., 2008) and not generally applicable. While both approaches are based on *reusing* existing technology, both work on the *contents* of each specification and are, thus, categorized as white-box solutions.

In contrast, pure black-box approaches do not intervene in the functionality of the reused transformation engine. Instead, they exploit the existing technology and the created artifacts during the transformation for propagating the annotations from the source to the target model *orthogonally* to the transformation.

Firstly, in (Buchmann and Greiner, 2018) this behavior is supported by using the provided DSL *MySync* allowing to specify corresponding elements in the source and target models. The single-variant transformation is executed as it stands and afterwards, the annotations are propagated to the corresponding elements as stated in the mappings. This approach works so far only for metamodels conforming to the Ecore meta-metamodel. Moreover, it requires the user to manually specify the corresponding elements for each new kind of input or output metamodel.

Another approach to achieve the behavior is trace-based propagation as proposed in (Westfechtel and Greiner, 2018). A generic trace model serves as interface for different kinds of traces. Annotations of source elements are applied to their corresponding target elements as recorded in the trace. It is a generic approach since it is independent of the applied transformation language and the input and output metamodel. The approach only requires a trace to be persisted. Therefore, in general it supports numerous transformation tools, like the ATL/EMFTVM (Wagelaar et al., 2012), medini QVT (ikv++ technologies, 2018) or QVT-d (Willink, 2017), Bxtend (Buchmann, 2018) and eMoflon (Leblebici et al., 2014).

Please note: Traces vary with respect to the granularity of the persisted information. As stated in (Westfechtel and Greiner, 2018) at least three categories can be distinguished: *incomplete*, *generation-complete* and *complete* traces. The first category only stores one source and one target element in so called *correspondence graphs*, being the key element in triple graph grammars (TGGs) (Schürr, 1994). In these realizations further dependent elements are determined from the basic mappings. In contrast, generation-complete traces persist all source and all target elements of a rule application whereas complete traces distinguish the elements of the target model with respect to the fact whether they have already been present before applying a rule (context elements) or they have been created due to applying a rule. Depending on the granularity of the trace, the propagation of annotations may have to be adapted. The authors give a formal proof on commutativity in the case complete traces are used and the transformation rules adhere to the underlying computational model.

Quite differently, if the transformation language supports aspect-oriented programming (Kiczales et al., 1997), a generic aspect could be provided to transfer the annotations (Greiner and Westfechtel, 2018). The aspect should attach the annotation of the element triggering its execution to the created target element. In the cited approach this behavior is implemented for the Xpand language (Klatt, 2007) supporting M2T transformations only. Thus, annotations are integrated as preprocessor directives and products filtered by using a preprocessor. While the approach is specific to the language Xpand (language-dependent), it can be categorized as black-box approach since transformation rules are not analyzed.

2.2 Correctness Criteria

For evaluating the validity of the target models of the MVMTs, a commutativity criterion has been postulated, among others, in (Westfechtel and Greiner, 2018; Greiner et al., 2017). This criterion is used for evaluating the quality of the annotations of the target model. As seen in Figure 1, it is based on comparing the outcome on the level of application engineering, i.e., the products. After executing the MVMT (t_{mv}), at first, both, the annotated source model m_s and the annotated target model m_t, are filtered by the same feature configuration fc. The filtered target model m_t'' should equal the model m_t' created when transforming the filtered source model m_s' with the same single-variant transformation t_{sv} used in t_{mv}. This property needs to hold for all valid feature configurations. Then, the transformation is said to *commute*.

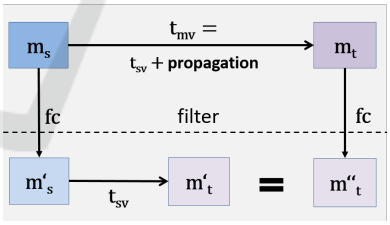

Figure 1: Commutativity criterion for evaluating the validity of the target model m_t of a multi-variant model transformation t_{mv} by filtering the superimposed models by all valid feature configurations fc.

To this end, the validity of the filtered target models is ensured in the following way: It is assumed the source model m_s is valid and correctly annotated such that each derived product m_s' is valid as well. Furthermore, with a valid model m_s' as input t_{sv} creates a valid target model m_t'. Since m_t'' should equal m_t', it is ensured that m_t'' is valid. If all filtered products m_t'' are valid, the superimposed target model m_t is semantically correct, too.

2.3 Error Measurement

For measuring the quality of the annotations attached to the target model m_t, an absolute error and one taking the affected model elements into account can be computed. In total, the error can be measured by counting the number of feature configurations in which commutativity is violated and by comparing it to the number of all valid feature configurations:

Definition 1 (Absolute Error). *Let n be the number of all valid feature configurations and let v be the number of feature configurations in which a strategy violates commutativity with $v \leq n$. Then, the absolute error err_{abs} can be computed in the following way:*

$$err_{abs} = \frac{v}{n}$$

This error rate is rather rigorous since it counts a feature configuration as wrong as soon as there is one difference between the models m_t' and m_t''. For that reason, the error could be relaxed by considering the number of differences between m_t' and m_t'' and comparing it to the number of elements in the multi-variant target model m_t. Summing these error rates up and dividing them by the number of valid feature configurations gives a hint on the overall error in terms of the affected elements (severity of the error).

Definition 2. *Let n be the number of all valid feature configurations. Let $|m_t|$ be the cardinality of m_t, i.e., the number of annotated elements in the target model. Let further $diff$ be the number of differences between m_t' and m_t''. Then, the severity error err_{sev} is calculated as follows:*

$$err_{sev} = \frac{\sum_{i=1}^{n} (\frac{\#diff_i}{|m_t|})}{n}$$

Both error rates allow to infer the quality of the MVMT with respect to fulfilling commutativity.

2.4 Consequences for the Transformation Framework

As seen in Section 2.1 a wide variety of MVMT realization approaches exists but yet there is no common means to evaluate the quality of the outcome with respect to achieving commutativity. Thus, there is a strong need for a framework evaluating whether the approaches fulfill commutativity. Such framework should respect at least the following requirements:

1. **R1: Reuse-based Execution.** For supporting the various approaches for generating a multi-variant target model, the execution needs to be replaceable. The framework should provide an interface for triggering already existing transformations. This interface needs to completely abstract from the different kind of transformations that are possible, i.e., in- or out-place, uni- or bidirectional, batch or incremental transformations and of kinds of input and output models. Alongside, the propagation of the annotations needs to be regarded and interchangeable.
2. **R2: Genericity.** The following components of the framework are necessary to be generic allowing an interchange of the different MVMT approaches but also of the SPLE tools and their specific capabilities:
   - the representations of the input and output models
   - the single-variant transformation
   - the kind of the feature model
   - the filter working with the feature model, its configurations and the different kinds of representing annotations
   - the comparison mechanism for the target products m_t' and m_t''; as a consequence of having different kinds of model representations

   Consequently, not only different MVMT approaches can be evaluated but also heterogeneous SPLE tools may be supported.
3. **R3: Availability of Results.** The framework is supposed to provide the results to the user. In the case commutativity is violated, the errors should be made available. If the locations of the violations are visible, the user can fix them or fix the transformation which was misbehaving. Additionally, the error values can be used to compare the quality of the different approaches.

3 FRAMEWORK

This section describes the evaluation framework for MVMTs by giving an overview on the framework followed by details on its two main components and their possible realizations.

3.1 Architectural Overview

First of all, as depicted in Figure 2 the framework is composed of two main parts: The *executor* (gray box) performs the multi-variant model transformation. It receives the multi-variant source model m_s and the reused single-variant transformation t_{sv} in order to

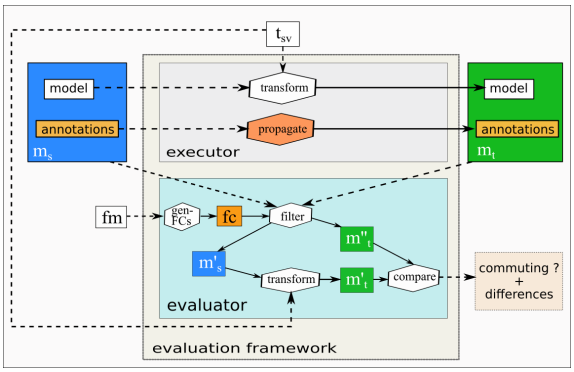

Figure 2: Schematic overview on the realization of the evaluation framework.

create the annotated target model m_t. The *evaluator* (light blue box), the second part, subsequently verifies whether a commuting model transformation is achieved with the target model created by the executor. Next, both parts are illustrated in greater detail. The most important generic interfaces and classes are captured in Figure 3. It highlights the relationships between the elements and shows which parts of the framework are replaceable to evaluate different MVMT approaches. Please note: both, the executor and evaluator are realized orthogonally. For that reason, it is possible to run the execution or the evaluation only.

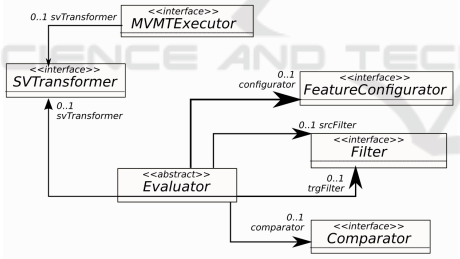

Figure 3: Simplified overview on the main exchangeable interfaces and abstract classes of the framework.

3.2 MVMT Execution

The task of the executor is to run the reused single-variant transformation t_{sv} and to invoke the respective mechanism to propagate the annotations to the target model. Thus, the executor is an interface offering the two tasks depicted in Figure 2: The transformation and the propagation. For this reason, the executor expects an SVTransformer to create the target model. The propagation needs to be provided according to the respective approach in the executor realization. Next, we introduce one realization of the MVMTExecutor interface for trace-based propagations.

As mentioned above, one possibility for propagating annotations is a trace-based approach. We provide an implementation of the generic executor interface using traces for the propagation. It uses the generic trace model proposed in (Westfechtel and Greiner, 2018) and thus, requires a replaceable converter to convert an arbitrary trace resource into aforementioned general trace model.

Then, the trace-based MVMT can take place: In the first step, the provided SVTransformer executes the reused transformation specification. Based on the resulting trace model, in the second part of the execution, annotations of the source are attached to the target elements. In the case a (generation-)complete trace was written, the annotations of all source and all context elements are combined in a conjunction and attached to all target elements. In the case of incomplete traces, only the annotations of 1:1 mappings are propagated to the single target element, leaving the target model partially without annotations.

One alternative realization using aspects (Greiner and Westfechtel, 2018) should be mentioned as well. It is meant to demonstrate the flexibility of the framework. Instead of the strict two-step procedure described above, the execution only consists of a single invocation of the workflow executing the single-variant transformation together with the aspect.

Quite alike, other MVMT approaches, like using a matching mechanism, like MySync, could be executed. Besides the multi-variant source model, only a transformer for the single-variant transformation needs to be provided and the propagation mechanism needs to be invoked.

3.3 MVMT Evaluation

After the execution of the MVMT, all remaining steps of the commutativity diagram are supported in the framework by the evaluator. The evaluation of commutativity consists of different steps which should be interchangeable. Therefore, all of the parts described next are generic interfaces.

Feature Configurations. At first, the feature model must be provided to a generic interface FeatureConfigurator. It generates all valid feature configurations with respect to the mandatory and optional features or further restrictions stated in the model. The feature model is a generic parameter of the interface for supporting different feature model types which commonly vary among the SPLE tools. So far, we provide a realization for generating all feature configurations for feature models conforming to the syntax of the Famile tool environment (Buchmann and Schwägerl, 2012) to the framework.

Filters. Given the feature configuration, the filter comes into play. The filter receives an annotated model and the feature configuration. It removes all elements from the model the annotation of which cannot be satisfied under the given configuration and persists the resulting single-variant model as one product of the SPL. In the framework the filter is as well a generic interface offering this task and, thus, allowing for providing different concrete realizations of the filter mechanism. While the filter receives the same feature configuration during the evaluation, the representations of the input models (source and target model) may vary, e.g., when the target model is a text representation.

One not regarding filters: In SPLE model filters can vary with respect to their capabilities to ensure consistent filtered products. Two kinds of model filters can be distinguished: a *flat* filter, always removing the elements where the annotation evaluates to false, and a *hierarchical* filter which takes dependencies inside the model into account and propagates selection states accordingly. For instance, in a spanning containment tree a hierarchical filter will remove children from the product if their parent is not included in the configuration. Furthermore, filters can vary on how to handle model elements missing an annotation. Besides others, one strategy includes such elements in all configurations regardless of the filter.

Moreover, if the target of the transformation is text instead of a model filter, typically a corresponding preprocessor will be needed to remove deselected text fragments. Then, the feature configuration must be turned into a preprocessor flag file and the target is a new source code project.

Comparison. Before finally comparing the two single-variant target models, the filtered source model needs to be transformed. Here, the same single-variant model transformation is used as the one generating the multi-variant target model. Thereafter, it is possible to compare both target models m_t' and m_t'' by using a comparator. The Comparator is a generic interface allowing for different comparison mechanism, e.g., for comparing two models or simply two strings, when the models are represented as text. Accordingly, the outcome may vary as well, e.g., being a difference model or a string when comparing text files.

For the comparison of two models our framework includes an implementation of the generic Comparator interface making use of the *EMFCompare* framework (Brun and Pierantonio, 2008). The result of comparing two models is a difference model showing matching and mismatching elements. Furthermore, it provides the total number of differences between the two models. The framework persists the difference model for the given feature configuration and uses the number of differences to compute the absolute and the severity error. The latter values are finally written to a file including the overall statistics values.

Evaluator. To this end, the framework provides an abstract evaluator executing all the single steps in one method as described in Algorithm 1. At first, for a given feature model all valid feature configurations are created and persisted. Then, all products are filtered by the two filters for the multi-variant source and target model. Then, the source products are transformed by the single-variant transformer. The subsequent comparison is executed by an instance of the Comparator which allows for different kinds of input models and comparison results as mentioned above. The resulting comparison data and the computed errors both can be written to files after the evaluation.

On the whole, the required components postulated in Section 2.4 are fulfilled in the following way:

- **R1: Reuse Transformation.** For the multi-variant as well as the single-variant transformations we provide an interface SVTransformer. It offers a single method transform allowing to specify the direction and whether the transformation should be executed in batch or incremental mode. The second part of the executor, the propagation, can be defined for each MVMT approach and can be invoked in the realization of the executor interface.
- **R2: Genericity.** First of all, no assumptions on the kind of in- and output models are made allowing for exchangeable representations. By further using the following interfaces and parameters the framework supports the replacement by components specific to the respective MVMT approach:
  - an interface SVTransformer the realization of which invokes the respective reused single-variant transformation
  - a generic parameter for the feature model for allowing different representations
  - a generic interface FeatureConfiguratior where based on the type of feature model all valid feature configurations can be created
  - a generic interface Filter receiving the multi-variant model (or source code), the feature configuration and the path for the filtered product
  - a generic interface Comparator allowing for changing the comparison method and the respective outcome

Algorithm 1: Process for evaluating commutativity.

```
1: procedure EVALUATE(fm, m_s, m_t)
2:     in fm                                          ▷ feature model as generic parameter
3:     in m_s, m_t                  ▷ multi-variant source and target model as generic parameter
4:
5:     List fcs = configurator .generateAllValidFC(fm)        ▷ creating all feature configurations
6:     List filteredSrc = srcfilter.filterProducts(fcs, m_s)    ▷ filter m_s by all configurations
7:     List filteredTrg = trgfilter.filterProducts(fcs, m_t)    ▷ filter m_t by all configurations
8:     List transfTrg = svTransformer.transform(filteredSrc)    ▷ transform all source products
9:     comparator.compareProducts(filteredTrg, transfTrg)   ▷ Compares the filtered target products
   with the corresponding transformed ones
```

All of the interfaces are provided to the generic evaluator which finally uses them in the overall evaluation.

- **R3: Availability of Results.** The output is written to files, stating the differences and the measured error values.

4 EXAMPLE

The goal of the evaluation framework is to verify whether an MVMT commutes. As mentioned before different approaches exist to implement MVMTs. In the following we provide an example in which the framework is used to verify the correctness of a trace-based propagation. The example is intended to demonstrate the behavior and functionality of the framework and to show how the single parts work. Due to space and illustration reasons we focus on describing one real world use case demonstrating the evaluation framework in action.

4.1 UML2Java Use Case

First of all, our scenario is a trace-based MVMT transforming an UML class diagram into the MoDisco Java model (Bruneliere et al., 2010). From the resulting model MoDisco generates corresponding Java source code. This is a necessary task to be performed during the model-driven development of a product line in order to implement its structural parts.

Our test scenario is centered around developing a Graph product line as described in (Lopez-Herrejon and Batory, 2001). The feature model – as presented on the left hand side of Figure 4 – consists of the mandatory features GraphProductLine, Nodes and Edges. A Search mechanism is optional to the Graph being exclusively either BFS or DFS. In addition, one or more of the proposed Algorithms can be selected. Nodes can be Colored and edges may be Weighted or Directed or both.

The corresponding UML class diagram (domain model) consists of classes and associations in order to realize the Graph product line. In total, it comprises 139 model elements including operations and properties. The annotated model is depicted in the tree representation of the Ecore editor on the left of the MVMT transformation in Figure 4. In the editor the annotation is placed behind the respective element. There are the mandatory classes Graph, Node, Edges and the optional classes Color, Search, Algorithm and Cycle, each annotated with the corresponding features. The class Adjacency is used when a search mechanism is selected and, thus, annotated correspondingly. Regarding the relationships between the classes, it must be noted that edges and nodes are connected either with one association in the case *undirected* graphs are desired (edgesToNodes) or with two associations, outEdgesToTarget and inEdgesToSource, for recording the target and source node or the outgoing and incoming edges, respectively. The associations are annotated accordingly. While the search mechanisms are realized as operations, the Weighted feature is respected with a property in the class Edge. Properties and operations are hidden in Figure 4 due to space reasons.

The single-variant transformation creating a Java model from the UML class diagram is a reused BXtend transformation as described in (Buchmann and Greiner, 2016). The transformation creates for each UML association a class declaration in the Java model capturing the two association ends as field declarations. For each UML class a Java class declaration is created and a compilation unit which is necessary to represent the corresponding Java file that is created by the MoDisco source code generation. Moreover, a parameterized type is created for each UML classifier in order to represent types with multiplicities greater than one. The Bxtend trace however, records only main correspondences, i. e., for example the UML

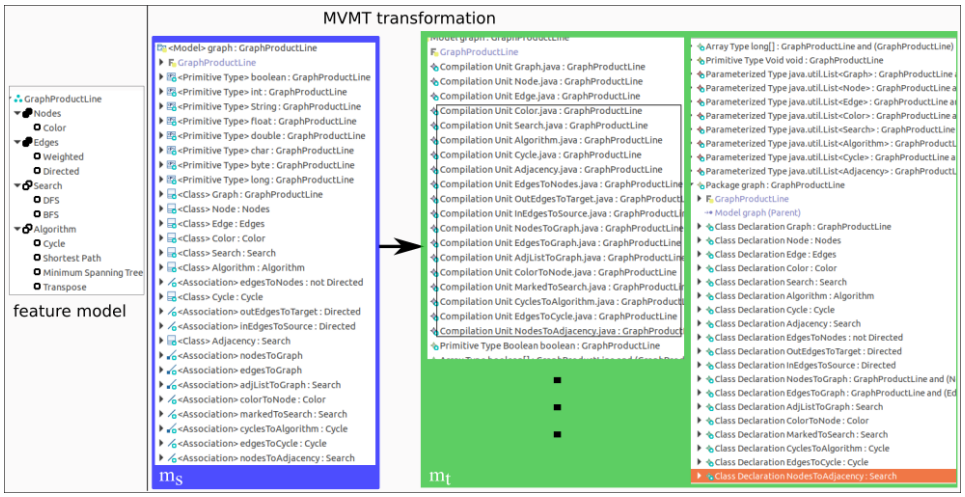

Figure 4: Transforming an annotated UML class diagram (left) into a multi-variant Java source code model. The annotations are boolean expressions over the featurs stated in the feature model on the left.

class and the Java class declaration or the UML association and the created class declaration. Additional created elements are left from the trace.

In our framework we can execute this transformation by using an executor for trace-based approaches. The single-variant transformer is a BXtendTransformer invoking the respective BXtend transformation and providing access to its trace. As mentioned in Section 2.1, traces vary with respect to their contents. BXtend only stores 1:1 mappings in a correspondence graph. For that reason, after executing the transformation, the trace propagation starts by assigning the annotations of the single source element to the single target element as stated in the trace. For example, a class declaration receives the annotation of the UML class. However, a certain number of elements in the target model remains expecting an annotation, like the corresponding compilation units or parameterized types. Their annotations are assigned by applying the most promising propagation strategy proposed in (Greiner and Westfechtel,): In a partially annotated model the annotation of the container is combined with the one of the contained elements and assigned. The resulting annotated target model is depicted in its tree representation on the right side of Figure 4: The corresponding elements mentioned in the trace are assigned the correct annotations. The compilation units and parameterized types receive the annotation of their container, which is the Java Model element.

In the subsequent step the evaluator is run. Here, we employ an implementation where the generic parameters are set to the kind of feature models and filter present in the SPL tool Famile (Buchmann and Schwägerl, 2012). The source and target model are considered to be stored in Ecore resources. Finally, the provided comparator triggers a model comparison with the EMFCompare framework (Brun and Pierantonio, 2008). While this comparison framework offers a number of strategies for differencing two models, e. g., by using UUIDs, we invoke the basic mode of matching properties of the model elements. As mentioned in Section 3.3 the result of this comparison is a difference model along with the number of differences. Counting the differences the total and the severity error are computed and finally written to a statistics file.

Running the evaluation in above scenario, first, a feature configurator for Famile feature models creates all valid feature configurations for the Graph feature model, i. e., 180 configurations. Thereafter, the multi-variant source and target model and the configurations are input to a filter invoking the Famile filter. Please note: We employ the default filter of the Famile tool, i.e., a hierarchical one which propagates selection states. Next, the source products are transformed by the same BXtendTransformer having been used in the MVMT execution. Finally, the model comparator tries to find matches between the transformed target products and the filtered target products. As last step, the absolute and the severity errors are computed and persisted to a difference file together with statistic values for all feature configurations.

Figure 5 presents excerpts of the resulting files in our scenario. The statistics file (diff.csv) lists in the first column the number of the feature configuration and in the second column the number of differences and states the error rates on the bottom. While the severity error is already high (44.8%), the absolute error reveals there is at least one difference in each

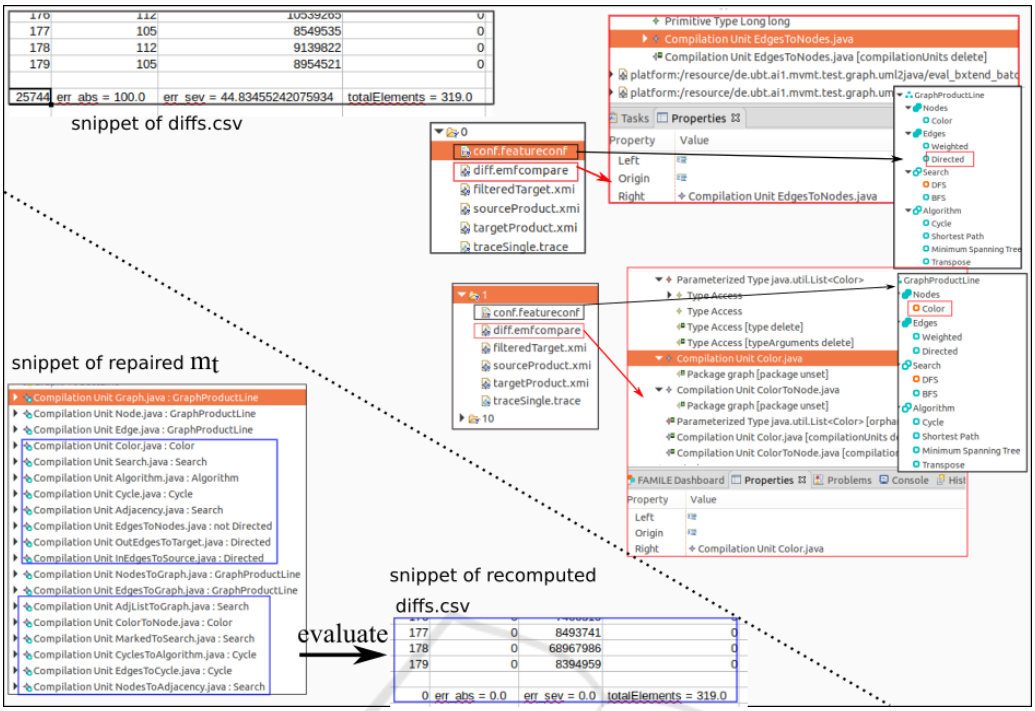

Figure 5: Results of first evaluation run (upper right) and after fixing the obvious errors (lower left).

product.

Having a look at the difference models of the first two configurations only reveals that the compilation units for the class declarations edgesToNodes, inEdgesToSource and outEdgesToTarget are present in the filtered target product (m_t'') but have no corresponding element in the left model of the comparison, i. e., in the transformed product (m_t'). Likewise, in the second configuration the compilation unit and the parameterized type for the class Color are included in the filtered target although the feature is not selected.

Both errors and similar ones are caused by the trace-based MVMT transformation that needs to find annotations for elements not mentioned in the trace. In the BXtend trace for the UML class, only the created class declaration is recorded and all additionally created elements, like the compilation units are not. Thus, they receive their annotation in a post-processing step. The annotation is composed of the annotation of its container and its contained elements. Since there are no contained elements for compilation units, the attached annotation is the one of its container, the Java model (GraphProductLine), which is a mandatory feature.

By only looking at the two configurations we can explain why we have a mismatch in each product in the example. In our product line there are two elements that are mutually exclusive. Either an edge connects two nodes without further information, in the case of an undirected graph or it records the source and the target node in two separate associations which are resembled by two classes and corresponding compilation units in the target model. Accordingly, annotating the compilation units with the root feature leads to an error when a directed graph is selected and the compilation unit for the edgesToNodes class is included and vice versa in the case of an undirected graph when the two specific compilation units for recording source and target nodes are part of the product.

Since the differences are made available, we can fix the obvious errors manually. In our scenario we need to change the annotation of the compilation units and parameterized types corresponding with abstract type declarations being annotated with an optional feature, e. g., those of the compilations units of the class Color or of the association classes edgesToNodes and its two partners. In total, a number of **18 elements out of the 319 elements** need to receive the more fine-granular annotation as seen in snippets on the bottom of Figure 5, e. g, the compilation unit for the class color is annotated with the feature Color.

With our evaluation framework it is possible to run the evaluation again without executing the transformation beforehand. Running the evaluation with the source and the modified target model, leads to 100% commutativity where no differences at all are found

between the transformed and the filtered target products. On the bottom of Figure 5 the resulting difference file is shown with the computed error rates.

5 DISCUSSION

Summing it up, as illustrated by the example our provided framework allows for evaluating the commutativity criterion for MVMTs. From above descriptions the main benefits become obvious: First of all, by making the differences available, erroneous behavior in the transformation may be detected. This allows the MVMT developer to fix errors in the annotated target model. Furthermore, instead of *fixing* errors, the developer can draw conclusions on *improving* the propagation approach. Moreover, the limitations of propagation approaches are detected. The scenario shows that with above approach automation may become impossible in real-world applications. Nonetheless, the example also reveals that errors may be fixed with a very small amount of effort (only 0.05% of all annotations need to be changed).

On its downside, when transformations are not commuting, it may become necessary to correct the annotations manually as a quick fix. In future work the task to find the correct annotation for a wrongly annotated model element could be automated. For example, the errors could be analyzed or compared to the corresponding feature configuration and the ones where the same model element is not causing a problem.

Although not demonstrated in the example but mentioned in the framework overview, most importantly the framework allows to evaluate the commutativity for different MVMT approaches by providing replaceable components for the single evaluation steps: The framework abstracts from the kind of feature models, the filters and their outcome. The input and output models do not have to conform to the Ecore metamodel but could also be String or other representations. Moreover, the comparison is generalized for different model representations and for the comparison outcome.

One note regarding scalability: Although the demonstrated example is a real-world transformation, on instance level it is still relatively small in terms of the number of annotated model elements and of optional features. In the case of product lines with a large number of optional features the brute force test requiring the generation of all feature configurations may become very expensive. Then, it is questionable whether all valid feature configuration can still be created in a feasible amount of time. However, the mechanism how to create feature configurations is also exchangeable and could be adapted, e.g., by using sampling or optimized satisfiability solvers for finding the valid feature configurations.

6 RELATED WORK

Transforming product lines has gained popularity in the last few years. Thereby, the term "multi-variant model transformation" may be interpreted differently: *transformation* of *multi-variant models*, the focus of our work, and *multi-variant transformation* of *models*, as addressed in (Strüber and Schulz, 2016; Strüber et al., 2018). In addition, *transformations of product lines* (Taentzer et al., 2017), covering both feature and domain models, goes beyond the scope of our work in which we evaluate the transformation of artifacts in a single software product line. Moreover, the tool presented in (de Lara et al., 2018) allows for summarizing families of (slightly varying) transformations into a product line which is not the focus of our work.

In contrast, our generic evaluation engine allows for transforming one kind of model representing a product line into another (also necessary) representation in the same product line automatically and for validating the resulting annotated target model. Since we seek for generic approaches to extend an SVMT to an MVMT, the framework abstracts from the reused single-variant transformation, the way of how features are provided and how the multi-variant models are filtered and finally, of how the target products are compared.

A number of different strategies to transform a multi-variant source model into a multi-variant target model have already been discussed in Section 2.1. To the best of our knowledge, so far there is no common means to evaluate the quality of the resulting target models. In addition, solutions which are formally proven (Westfechtel and Greiner, 2018; Taentzer et al., 2018; Strüber et al., 2018) ensure the transformation behaves correctly, if the underlying computational model is satisfied. However, if the computational model is violated, our framework helps to find out where the transformation fails and why in order to improve the MVMT solution for supporting more (general) use cases.

7 CONCLUSION

All in all, this paper introduces a generic evaluation framework for multi-variant model transformations

generating an annotated target model from a given annotated source model in an SPLE scenario. It allows for reusing arbitrary single-variant transformations and defining the propagation of annotations in its execution component. The following evaluation abstracts from all remaining evaluation steps. It is possible to interchange the kind of feature model and thus, how all feature configurations are determined, as well as how the products are filtered and finally compared.

To the best of our knowledge, it is the first realization not specifically trimmed for evaluating one approach but abstracting from different possibilities to create the target model. In this way it offers the possibility to evaluate different reuse-based MVMT proposals. Furthermore, by making violations to commutativity available to the user, the user is able to fix erroneous parts in the resulting model or to improve the transformation approach.

In the future the framework can be used to conduct large case studies on different MVMT approaches based on varying (real-world) transformation scenarios. The findings may provide deeper insights in the strengths and weaknesses of the single solutions and allow to find optimizations.

REFERENCES

(2017). *Unified Modeling Language (UML)*. Object Management Group, Needham, MA, formal/17-12-05 edition.

Apel, S., Janda, F., Trujillo, S., and Kästner, C. (2009). Model Superimposition in Software Product Lines. In *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*, pages 4–19.

Brun, C. and Pierantonio, A. (2008). Model differences in the eclipse modelling framework. *UPGRADE*, IX(2):29–34.

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 '10, pages 173–174, New York, NY, USA. ACM.

Buchmann, T. (2018). BXtend - A Framework for (Bidirectional) Incremental Model Transformations. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 336–345.

Buchmann, T. and Greiner, S. (2016). Bidirectional model transformations using a handcrafted triple graph transformation system. In *Software Technologies, 11th International Joint Conference, ICSOFT 2016, Lisbon, Portugal, July 24-26, 2016, Revised Selected Papers.*, pages 201–220.

Buchmann, T. and Greiner, S. (2018). Managing Variability in Models and Derived Artefacts in Model-driven Software Product Lines. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 326–335.

Buchmann, T. and Schwägerl, F. (2012). FAMILE: tool support for evolving model-driven product lines. In *Joint Proc. co-located Events at 8th ECMFA*, CEUR WS, pages 59–62, Lyngby, Denmark.

Czarnecki, K., Antkiewicz, M., Kim, C. H. P., Lau, S., and Pietroszek, K. (2005). Model-driven software product lines. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 126–127.

de Lara, J., Guerra, E., Chechik, M., and Salay, R. (2018). Model transformation product lines. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 67–77.

Famelis, M., Lucio, L., Selim, G. M. K., Sandro, A. D., Salay, R., Chechik, M., Cordy, J. R., Dingel, J., Vangheluwe, H., and Ramesh, S. (2015). Migrating automotive product lines: A case study. In *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 82–97.

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.

Greiner, S. and Westfechtel, B. On determining variability annotations in partially annotated models. In *Proceedings of the Thirteenth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2019, Leuven, Belgium, February 6-8, 2019*. (in press).

Greiner, S. and Westfechtel, B. (2018). Generating multi-variant java source code using generic aspects. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 36–47.

Heidenreich, F., Kopcsek, J., and Wende, C. (2008). FeatureMapper: Mapping features to models. In *Companion Proc. 30th ICSE*, pages 943–944, Leipzig, Germany. ACM.

ikv++ technologies (2018). *medini QVT*. ikv++ technologies. http://projects.ikv.de/qvt.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008).

ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39.

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.

Klatt, B. (2007). Xpand: A closer look at the model2text transformation language. *Language*, 10(16):2008.

Leblebici, E., Anjorin, A., and Schürr, A. (2014). Developing emoflon with emoflon. In *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, pages 138–145.

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.

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.

Schürr, A. (1994). Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, pages 151–163.

Schwägerl, F., Buchmann, T., and Westfechtel, B. (2016). Multi-variant model transformations - A problem statement. In *ENASE 2016 - Proceedings of the 11th International Conference on Evaluation of Novel Approaches to Software Engineering, Rome, Italy 27-28 April, 2016.*, pages 203–209.

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

Stahl, T., Völter, M., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-driven software development - technology, engineering, management*. Pitman.

Strüber, D., Peldszus, S., and Jürjens, J. (2018). Taming Multi-Variability of Software Product Line Transformations. In *Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings.*, pages 337–355.

Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., and Plöger, J. (2018). Variability-based model transformation: formal foundation and application. *Formal Aspects of Computing*, 30(1):133–162.

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

Taentzer, G., Salay, R., Strüber, D., and Chechik, M. (2017). Transformations of software product lines: A generalizing framework based on category theory. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, pages 101–111.

Taentzer, G., Salay, R., Strüber, D., and Chechik, M. (2018). Transformation of software product lines. In Tichy, M., Bodden, E., Kuhrmann, M., Wagner, S., and Steghfer, J.-P., editors, *Software Engineering und Software Management 2018*, pages 51–52, Bonn. Gesellschaft fr Informatik.

Wagelaar, D., Iovino, L., Ruscio, D. D., and Pierantonio, A. (2012). Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In *Theory and Practice of Model Transformations - 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings*, pages 192–207.

Westfechtel, B. and Greiner, S. (2018). From single- to multi-variant model transformations: Trace-based propagation of variability annotations. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 46–56.

Willink, E. D. (2017). The micromapping model of computation; the foundation for optimized execution of eclipse qvtc/qvtr/umlx. In *Theory and Practice of Model Transformation - 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings*, pages 51–65.