On Preserving Variability Consistency in Multiple Models

Sandra Greiner
Applied Computer Science I – University of Bayreuth
Bayreuth, Germany
Sandra1.Greiner@uni-bayreuth.de

Bernhard Westfechtel
Applied Computer Science I – University of Bayreuth
Bayreuth, Germany
bernhard.westfechtel@uni-bayreuth.de

ABSTRACT

Model-driven software product line engineering (MDPLE) is a holistic approach to realize variability-intensive systems by using models. In MDPLE the usage of models aims to increase the level of automation by reducing the product derivation to a pure code derivation step. Since models are present at different development phases, they have to be kept consistent all over these phases, for example by storing information about corresponding elements in model transformations. Reasons why to use model transformations or similar automated mechanisms are manifold. For instance, if the product line is built in a forward-engineering process, model transformations will be beneficial to propagate the coarse-grained information of an early phase to the subsequent phase automatically. In contrast to single-variant engineering, in MDPLE there is not only the challenge to keep multiple models consistent but also their presence conditions. Since variability mechanisms and the ways how presence conditions across different models are maintained vary, this contribution categorizes the consistency maintenance of presence conditions in MDPLE approaches to give an overview of already existing techniques. As a result, we find that while several automated solutions to keep presence conditions across models consistent exist, they are not employed in the MDPLE tool landscape.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software product lines**; Abstraction, modeling and modularity.

KEYWORDS

Model-driven Software Product Line Engineering, multi-variant model transformations, software evolution, multi-view modeling

ACM Reference Format:
Sandra Greiner and Bernhard Westfechtel. 2021. On Preserving Variability Consistency in Multiple Models. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21), February 9–11, 2021, Krems, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3442391.3442399

VaMoS'21, February 9–11, 2021, Krems, Austria

ACM ISBN 978-1-4503-8824-5/21/02…$15.00
https://doi.org/10.1145/3442391.3442399

1 INTRODUCTION

Software Product Line Engineering (**SPLE**) is a software engineering discipline to support mass customization by relying on *variability* and *organized reuse*. Commonalities and differences of the software system to be built are expressed in form of *features* captured in variability models, such as feature [28] or decision [40] models. A complete selection of the features in a *feature configuration* is used to derive a customized product (i.e., one variant of the system). To build product lines, three major variability mechanisms stand out: *compositional* [5], *transformational* (also known as delta-oriented) [39] and *annotative approaches* [16], where the first two mechanisms are associated with *positive variability*, i.e., a single-variant (core) product is extended or modified to become a new product, and the annotative approach with *negative variability*, i.e., from a *superimposition* of all products, elements corresponding to deselected features are removed [2].

The combination of SPLE with *Model-Driven Software Engineering* (**MDSE**) [7, 46], called *Model-Driven Product Line Engineering* (**MDPLE**), [13] promises to increase the level of productivity due to automation. On the one hand, MDSE grounds on *models* which are instances of *metamodels* and appear at all development stages (analysis, design, implementation) at different levels of granularity. To this end, executable code is generated completely automatically from these models. On the other hand, the enabling technology in MDSE are *model transformations*, allowing to transform a model into another (model) representation. However, in MDPLE, models have to be lifted from the single-variant to the multi-variant context, i.e., such *domain models* represent variable contents at the level of domain engineering. In annotative approaches, a *multi-variant* (also superimposed) model reflects the entire variety of products in the product line, i.e., each variant. *Presence conditions*, to which we refer in the sequel as *annotations*, specify a Boolean expression over features and have to be associated with domain model elements to declare in which configurations these elements are visible. Annotations are not only present in annotative approaches but the *variability traceability* [2, 6] of features to artifacts is essential in compositional and transformational approaches as well.

As mentioned above, the product line typically consists of various domain models, such as design (e.g., UML or Ecore class diagrams) and implementation models (e.g., a Java model [8, 25]) which all have to carry annotations for deriving products. However, it is a laborious and error-prone task to assign annotations to each model manually anew, contradicting the increase in productivity and automation. As a consequence, *multi-variant model transformations* [19, 38, 50] allow for propagating annotations from one model to the other, thus, keeping the annotations of the models consistent. However, while there are many techniques to automate the propagation of annotations, they are rarely employed in product line tools and approaches. For that reason, in this paper we inspect an

exemplary set of MDPLE tools and approaches to examine how they maintain annotations and propagate them across models in a product line. For a proper comparison we contribute classification criteria to distinguish the MDPLE approaches with respect to their maintenance of annotations and how they keep them consistent. We find that only one tool applies an automated propagation approach, there are few approaches that do not require a propagation and, most importantly, none employs the consistency criterion of commutativity which is guaranteed to be satisfied by some of the multi-variant model transformation solutions.

2 MOTIVATION

This section introduces an example scenario to demonstrate the usage of multiple models and keeping their annotations consistent.

2.1 Example Scenario

Figure 1 sketches a small product line for database contents. The feature diagram consists of a *mandatory* feature `Person` and an *optional* feature `Family`. Thus, the database "products" always store persons but not always relate persons into families. A corresponding UML [33] class diagram is modeled in the design phase as domain model. For deriving source code, an additional Java model (e.g., **MoDisco** [8] or **JaMoPP** [25] model) is maintained which can be created by transforming the UML class diagram either manually or automatically in a model transformation. The example involves the following four steps:

(1) *Create UML class diagram*
(2) *Annotate UML class diagram*
(3) *Create corresponding Java model by a model-to-model transformation*
(4) *Annotate Java model*

In the first step, the UML class diagram is created. The model comprises classes for the database, persons and families, thus, it is a *multi-variant model* where all variants of the product line are cumulated. A person always carries a name (derived property) which is composed of up to three first names followed by one last name. Families are also named. As further shown in Figure 1, we exemplify here an annotative variability mechanism: In the (already performed) second step the model is manually annotated so that elements carry annotations to declare their visibility in different products. For instance, the last name of a person is only present if families are not included in the product (annotation: `!Family`). During the product derivation elements are removed, the annotation of which is not satisfied by the given feature configuration.

For easier synchronizing source code changes with the design model, in the third step the UML class diagram is transformed into a Java model by a model transformation. The Java model offers the benefit to exploit existing tool support that keeps the Java model and the generated source code consistent [8]. To keep this example small and understandable we do not go into all details of a UML to Java transformation [18] but apply the following mappings. For each UML class a Java `ClassDecl(aration)` and `CompilationUnit` are created. Compilation units are stored in the overall container, the Java model, whereas classes are part of their corresponding package. Furthermore, UML properties yield the creation of `FielDecl(arations)` which are stored in the corresponding class declaration. Please note: we assume here that a state-of-the-art (single-variant) transformation is employed which is typically unaware of the variability annotations present in multi-variant models. Moreover, the example abstracts from the way annotations are attached to model elements, which typically varies in the MDPLE approaches (c.f. subsection 3.2).

In the last step, corresponding annotations should be assigned to the Java model to allow for deriving products from it. These annotations must be assigned to each model element in the Java model according to the annotation of its corresponding UML element. Generally, without dedicated techniques annotations will have to be provided to every model of the product line *manually* if tool support is missing. Annotating each model, however, is a laborious, repetitive task which provokes errors. Therefore, an automation, such as a transformation being able to propagate annotations in addition to creating the target model, is indispensable. *Multi-variant model transformations* (c.f. subsection 4.2), which are mainly proposed for annotative approaches, are one way to solve this task.

Since annotative approaches are not the only means to build a model-driven product line from multiple metamodel instances, we raise two research questions:

- **R1:** How does the annotation maintenance of elements in a single model vary?
- **R2:** How are annotations kept consistent across multiple models?

Answering these questions allows for comparing the approaches more easily and strives for the possibility to integrate automated techniques in those MDPLE approaches, which hinder the increase in automation by manually maintaining annotations.

3 CLASSIFICATION PROPERTIES

This section presents different criteria to classify MDPLE approaches with the special focus on maintaining multiple models in a product line. Figure 2 organizes the criteria in a feature model. At first, the section discusses general properties with respect to the variability mechanisms followed by presenting the classification criteria.

3.1 General Notes

On a coarse-grained level, MDPLE approaches and tools can either realize positive or negative variability and be classified accordingly [42]. While compositional and transformational (delta-oriented) approaches realize positive variability, i.e., a core product is modified to become a new customized product, annotative approaches realize negative variability where elements are removed from a superimposition. Here we contribute criteria based on which each of the variability mechanisms can be further classified. Particularly, we concentrate on how the annotations of multiple models in one product line are kept consistent (independent of the variability mechanism). As a consequence, in the following we discern the MDPLE approaches by the way they map annotations onto model elements and deduce how they handle the maintenance of annotations during the product line life cycle.

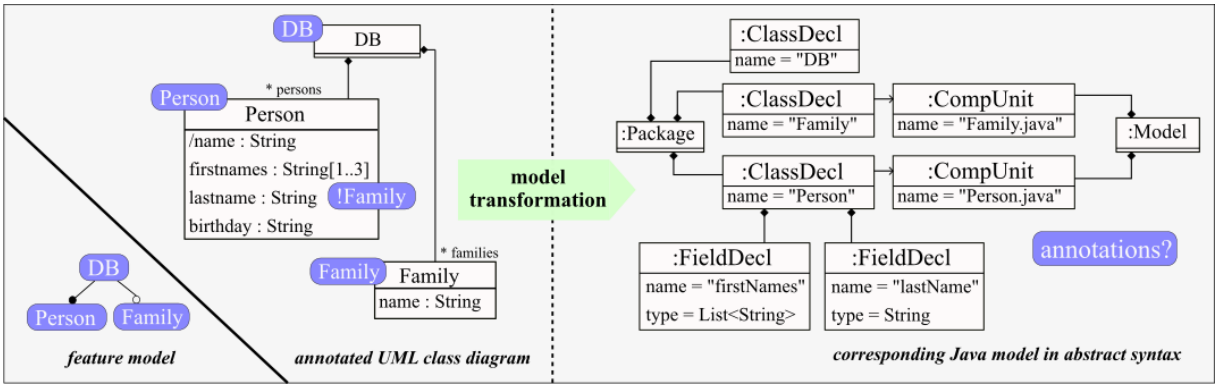

Figure 1: Usage of multiple (annotated) models in one product line.

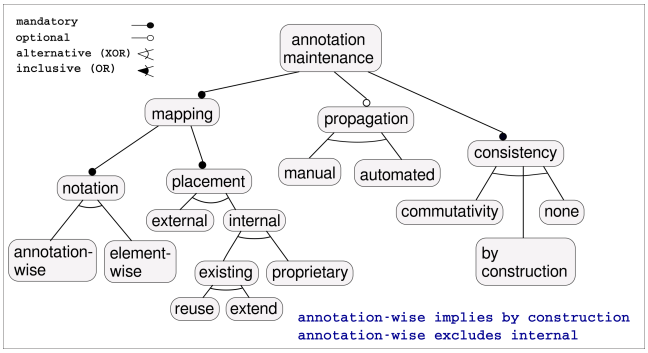

Figure 2: Classification properties for maintaining annotations.

3.2 Mapping

On the left-hand side, the feature model in Figure 2 mentions the *mapping* which is discerned with respect to the features *notation* and *placement*. The feature *mapping* corresponds with the SPLE property *variability traceability* of artifacts to features [2, 6]. In the present paper, the mapping feature serves the purpose to distinguish ways *how* annotations are associated with domain model elements.

3.2.1 Notation. In the first place the *notation* of the mapping which is either per element or per annotation must be distinguished because it implies whether an annotation propagation is necessary.

An *annotation-wise* mapping means that model elements are linked to the concrete feature or a variation point in a variability model which they are realizing. Consequently, these links are frequently stored in a dedicated *correspondence model* [1, 17] or other kind of file enumerating the features or Boolean expressions over them as exemplified on the left-hand side of Figure 3. This implies that there is exactly *one* annotation for each feature or expression over features onto which all corresponding model elements are mapped, regardless of the metamodel they are instance of. Such mechanism is present, for instance, when multi-view models are used [17]. Each view is an instance of a different metamodel and (at least) one view represents a variability model. Elements of the other views are linked by a correspondence model (another view) onto one variation point or a feature in a feature model.

On the upside, for the product derivation (and traceability) only the collection of annotations, e.g., the correspondence model, needs to be looked up. Moreover, it is not necessary to propagate annotations whenever a model element changes. Instead, new elements can be added in the correspondence model to a certain annotation and removing an annotation from an element requires to remove the model element from the correspondence model. Similarly, a change of an annotation is immediately available to all model elements. On the downside, complex annotations, such as propositional formulas, may only restrictively or complicatedly be supported. Similarly, more complex behavior such as the introduction of alternative values for a model element may be considered an inconsistency.

In contrast, element-wise annotations, as depicted in the middle and on the right side of Figure 3, may (but do not have to[1]) be assigned to each element appearing in a model. As a consequence, depending on the approach arbitrarily complex annotations can be associated with an element easily. Element-wise annotations strongly relate with the *placement* criterion, explained next.

Notice: In compositional and transformational approaches an entire module is annotated with an annotation. Such *module-wise* annotations are assigned to a summary of elements in a compositional approach or a collection of operations in a transformational approach. Since currently the annotation of a module is unique for a metamodel but must be mentioned again when an instance of another metamodel is modified by a new delta module, module-wise mappings are regarded as a special kind of element-wise mapping.

3.2.2 Placement. The *placement* feature signifies, whether the mapping is stored *internal*ly or *external*ly. While many approaches store the annotations internally (i.e., in the domain model(s) itself), often by exploiting the capabilities of the chosen metamodels, an external mapping approach separates the concerns of specifying annotations in (at least) one file and modeling the domain in another.

An *internal* mapping is either realized by exploiting the existing modeling language constructs, which can either be *reused* without modification or by *extending* the language. Alternatively, new *proprietary* product line (mapping) languages are developed.

[1] Missing annotations are frequently interpreted as omnipresent, i.e., as `true`.

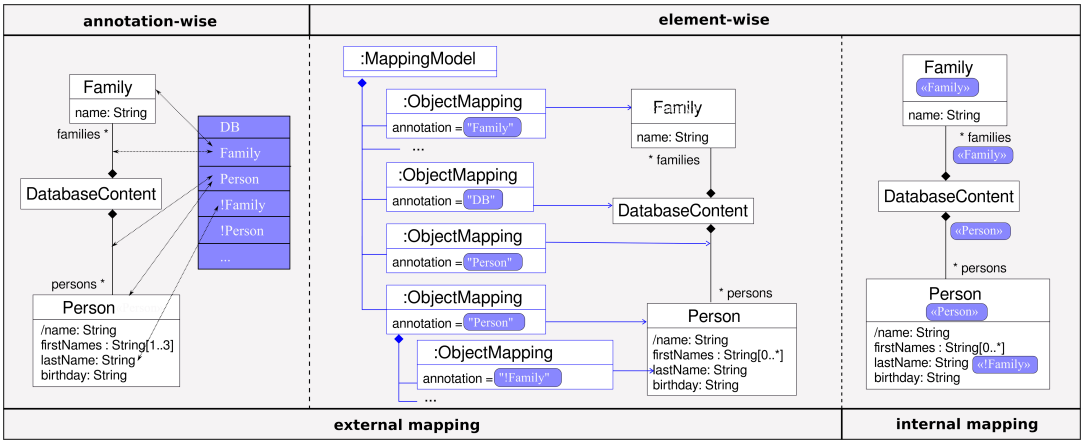

Figure 3: Forms of mapping annotations onto model elements.

The mapping sketched on the right hand-side of Figure 3 demonstrates an internal reuse mechanism where UML class diagram elements are annotated with customized stereotypes. Such approach, exploiting UML profiles, is proposed by Czarnecki and Antkiewicz [12]. Another approach for UML [16] also utilizes UML profiles but in a static way to mark elements as, for instance, `<<kernel>>` or `<<variant>>` elements. How to use static profiles, particularly for product derivation, to the best of our knowledge, was mentioned by Ziadi et al. [54] first.

Contrastingly, *proprietary* languages are designed for developing product lines in a holistic way, where the annotation is integrated in the multi-variant "model". As an example, the language **Clafer** is such product line modeling language which stores structural and behavioral elements, as well as the annotation, in small units denoted as *clafers* [3, 4, 27].

In case the internal mapping is not part of a newly developed language, the obvious benefit of internal mappings is the reuse of the existing language and, thus, of existing tool support. However, the reuse comes with two main shortcomings. First, it is *not generally* applicable because the metamodel must incorporate at least one element, such as an `EAnnotation` in Ecore models, in which annotations can be stored and in this way mapped onto model elements. Second, in case such mapping element is present, the *granularity* of assigning annotations depends on up to which granularity the modeling language is capable to associate the mapping element with its other elements. For instance, since in Ecore models structural features cannot be annotated with an `EAnnotation`, Reuling et al. [37] transform the Ecore models into a variational abstract syntax graph which integrates structural features as first-class entities allowing to annotate these elements as well. Due to intertwining variability and domain modeling, a general downside of internal mappings is the loss of separation of concerns.

In contrast, an external mapping builds on a separate mapping file, which typically maps model elements onto annotations. While provoking a higher maintenance effort and additional complexity, external mappings offer the advantage of being customizable and, thus, more powerful than an internal mapping (which is restricted by the capabilities of the language). Besides using external mappings in form of correspondence models for annotation-wise mappings, such approach is, for example, realized in the annotative tools **FeatureMapper** [26] and **Famile** [10] as element-wise mapping.

The middle of Figure 3 presents excerpts of an external mapping in abstract syntax in the way it is realized in **Famile**. Mapping models in this tool exactly reflect the hierarchical structure of the domain model they are used for. Each object mapping references the model element of the corresponding domain model and can store an annotation. This way of modeling allows for alternative mappings, such as renaming a class in case another feature is selected.

On the whole, annotation-wise mappings may be beneficial if few changes in the elements occur and the complexity of the annotations remains low as well as when it is used for different models. In this case, the annotation file can simply be changed without considering each element which carries the modified annotation. Contrastingly, element-wise mappings are beneficial if the annotations of single elements are complex and vary a lot among the elements. Moreover, internal mappings are a straightforward way to reuse existing modeling capabilities but on their downside imply a scattering of annotations. Thus, although adding additional complexity and maintenance effort, the upside of external mappings are the separation of concerns, a dedicated place to look up annotations and the possibility to customize the annotation capabilities.

3.3 Annotation Propagation

All other criteria classifying the propagation of annotations depend on the fact, whether annotations are stored element-wise or annotation-wise. In case of an element-wise mapping, a *propagation* of annotations from the annotated model to another is beneficial for avoiding the redundant and error-prone effort to assign annotations to corresponding elements manually. Consequently, assigning annotations manually resides at the lowest level of automation. More advanced approaches propagate annotations automatically.

The feature *automated* means that changing the annotation of a model element or creating a new model representation allows to automatically propagate annotations to corresponding elements.

When creating the Java model in our motivating example, an automated propagation means that the annotations of UML elements transfer to corresponding elements in the Java model without having to specify new rules. For example a *multi-variant model transformation* (c.f. subsection 4.2) can accomplish this task by executing a reused *(single-variant) model transformation* that creates the Java model and assigns annotations additionally.

3.4 Consistency

In SPLE different forms of consistency are relevant. Feature configurations must be consistent with the feature model (i.e., no constraints of the feature model should be violated) but also the derived products need to be well-formed (i.e., they need to conform at least syntactically to their syntax definition). For the maintenance of corresponding annotations in different models (created by model transformations), the *commutativity* criterion, sketched in Figure 4, which was, to the best of our knowledge, introduced by Salay et al. [38], is a key property for annotation consistency across models.

The product m'_s derived from a multi-variant source model mv_s, such as the UML class diagram in our example, results in a product m'_t when being transformed with a state-of-the-art single-variant model transformation t_{sv}. In contrast, the multi-variant model transformation t_{mv} creates an annotated target model mv_t, such as the Java model in our example. Deriving a product from mv_t must result in a product m''_t which is equivalent to the product m'_t resulting from applying the same feature configuration on the multi-variant source model and transforming it. If the diagram in Figure 4 commutes for each valid feature configuration, the transformation t_{mv}, particularly the propagation of annotations, is correct.

Alternatively, the assignment of corresponding annotations can be ensured *by construction* which is the case in annotation-wise mappings. In these approaches the construction process guarantees that corresponding elements are assigned the correct annotation because all elements are added manually to the right annotation. While commonly annotation-wise mappings are assigned in the first place manually, in incremental scenarios they can be updated automatically. The change of an annotation (e.g., when renaming a feature) automatically occurs in all corresponding elements in annotation-wise mappings whereas it has to be updated by handwork in *manual* solutions.

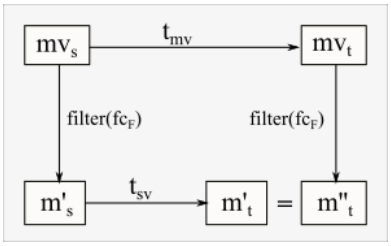

Figure 4: *Commutativity* of transformations as consistency criterion. Adapted from [22]

4 APPROACHES

This section considers an exemplary set of MDPLE tools and approaches and checks how they maintain the development of multiple models for a product line, particularly focusing on the maintenance of annotations. We select a broad, but not exhaustive set of tools and variability languages realizing one of the three variability mechanisms. At first, we regard tool-specific approaches followed by an examination of automated solutions, namely (multi-variant) transformations which are (mostly) designed in a MDPLE tool-independent way. We conclude with a discussion of the findings.

Please note: While many product line engineering processes [2, 11, 36] foster the usage of instances of different metamodels during development, they do not specify how to maintain annotations. Therefore, we do not take these processes into account.

4.1 Tool-Specific Approaches

Few approaches explicitly mention in their publication the maintenance of multiple models along different development stages. An exception are tools and methods which realize multi-view modeling of product lines. This idea is introduced by Gomaa and Shin [17] for UML models only and present, for instance, in **VaVe** [1], the product line extension of **Vitruvius** [31]. These approaches need to handle annotations of different models natively. Table 1 summarizes the categorization of all approaches illuminated in the following. Please note: The manual annotation assignment in element-wise mapping approaches always implies that consistency cannot be ensured.

At first, we look at compositional tools which follow the model-driven paradigm. *Aspect-oriented modeling* [53] is one means to realize positive variability in a compositional way. Model elements can be defined as join points and an *aspect* includes feature-specific extensions which are typically woven into the base model during a model transformation. The concepts and realization of aspect-oriented MDPLE (**AOMDPLE**) were published by Groher and Voelter [23] but are also implemented based on graph transformations in the tool **MATA** [52]. MATA supports aspects in UML sequence, state and class diagrams by extending the specification whereas the AOMDPLE approach by Groher and Voelter *reuse* languages of the openArchitectureWare[2] (now predominantly Xtext and Xtend) project as they stand. While these approaches place an *element-wise internal* mapping by defining join points, they do not perform a dedicated propagation of annotations across models. The *manually* assigned annotations in form of join points define which model element(s) are modified by executing an aspect incorporating the customized contents.

Secondly, delta-oriented tools, such as **DeltaEcore** [43, 44], **SiPL** [34, 35] and **DarwinSPL** [32] and transformational approaches, in general, employ a *core* model which is extended by *delta modules*. A delta module consists of *delta operations*, which modify the core model, and an *application condition* (i.e., an annotation). In our example, a core model of the UML class diagram may consist of the classes `DB` and Person. A delta module annotated with the feature `Family` may add the class `Family` and delete the lastname from the person class. Consequently, the annotation is assigned module-wise (i.e., element-wise) and can be considered to be *internal* because it is integral part of the delta module and not located in a separate

[2] http://www.eclipse.org/Xtext/

Table 1: MDPLE approaches, where the first third of the table mentions approaches realizing positive variability, the second part annotative variability mechanisms including a projectional mechanism to switch between representations. The last two rows present multi-view approaches.

Reference	short description	mapping notation	mapping placement	propagation	consistency
[23]	AOMDPLE	element-wise	internal, reuse	manual	none
[52]	**MATA**	element-wise	internal, extending	manual	none
[34, 43, 52]	delta-oriented tools	module-wise	internal, proprietary	manual	none
[15, 24]	**CVL**	element-wise	external	manual	none
[55]	**VML***	element-wise	external	manual	none
[12, 14]	template-based	element-wise	internal, reuse	manual	none
[3, 4, 27]	**Clafer**	element-wise	internal, proprietary	manual	none
[26]	**FeatureMapper**	element-wise	external	manual	none
[10]	**Famile**	element-wise	external	manual	none
[41]	**SuperMod**	element-wise	internal, proprietary	automated	none
[37]	projectional editor	element-wise	internal, proprietary	manual	none
[17]	multiple UML views	annotation-wise	external	manual	by construction
[1]	**VaVe**	annotation-wise	external	manual	by construction

file or model. Since in the published state of these tools a module is unique for a metamodel, modules, including their annotation, have to be created for each new metamodel *manually*.

The variability modeling language **CVL** (*Common Variability Language*) [15, 24], as well as the *Family of Variability Mapping Languages* (**VML***) [55], both support a positive variability mechanism by realizing external, element-wise mappings. The VML* metamodel allows to define pointcuts for exactly one target model based on a variability model. Since it only supports the creation of mapping models, a propagation of information of one mapping model to another is not foreseen. CVL follows a transformational mechanism where a base model and replacement libraries are created. The base model and replacement models are specific to one metamodel and require manual maintenance of annotations across multiple models.

In annotative approaches, annotations are also either stored internally or externally. One of the first techniques employs a model template [12, 14] which superimposes all variants of one metamodel (i.e., it is a multi-variant model in our terminology). The template instances are single products derived in a model-to-model transformation based on a feature configuration. The template incorporates *element-wise* annotations *internally* based on *reused* UML stereotypes defined in a variability profile. A propagation of corresponding annotations is not foreseen.

The *proprietary* language **Clafer**, which was mentioned in subsection 3.2, serves as an example where annotations are put *element-wise* (or module-wise) for each *clafer* which is a unit storing structural and behavioral elements as well as its annotations. An automated determination of annotations for a new clafer is, to the best of our knowledge, not considered.

Furthermore, the tools **FeatureMapper** [26] and **Famile** [10] both involve mapping models which relate elements of multi-variant domain models, conforming to the eMOF standard, to annotations which are Boolean expressions over the features of a feature model. The tools mainly differ in the granularity up to which annotations can be assigned to model elements. **Famile** allows for a very fine-grained mapping also of structural features, such as the name of a class, onto an annotation as well as for certain consistency mechanisms for deriving products from one model. Both employ element-wise external mappings and require to create a separate mapping model for each domain model manually. Thus, an automated propagation of annotations to corresponding elements is not supported[3].

Tools realizing filtered or projectional editing of models, internally maintain superimposed models. In **SuperMod** annotations are assigned element-wise as part of a superimposed model, which is hidden from the user, and automatically maintained upon commits. Annotations are part of the self-defined Ecore model representing the superimposed model which may store instances of different metamodels. We classify the superimposed model as a *proprietary* language. Although the maintenance of annotations upon commits takes place automatically and may involve elements of different models, it is unaware of dependencies between specific model elements. Consequently, consistency across models cannot be ensured by this form of propagation. Similarly, a projectional editor for model-driven product lines allows to switch between delta-oriented and annotative representations of the domain model [37]. Like in SuperMod, internally a multi-variant model (called *variational abstract syntax graph* (**VASG**)) is maintained where the annotations are stored element-wise in the multi-variant model itself, i.e., internally. This model is also classified as proprietary because it is designed specific to the tool, although the projection exposed to the developer is reusing Ecore models as they stand. Despite the fact, that representations of the model can be changed and, thus, have to be transformed, annotations are not propagated across models. The only annotation "completion" takes place in case VASG elements are missing one triggering an automated determination of the missing annotation.

Contrastingly, multi-view approaches [1, 17] keep multiple models consistent upon modifications applied to one model. Gomaa and Shin [17] refer to each domain model, such as activity and

[3] Research on multi-variant transformations [19, 22] is implemented for the tool Famile.

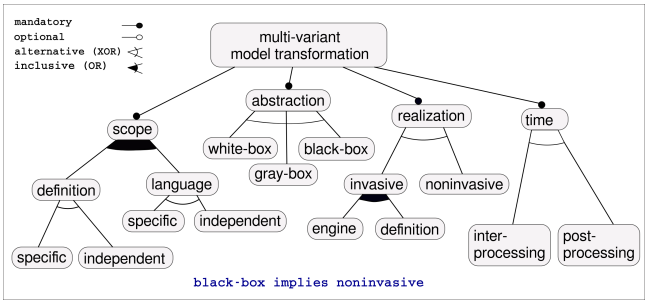

Figure 5: Classification properties for multi-variant model transformations. Adapted from [51].

class diagrams, as a *view*. Besides other views, a correspondence model is put in between the views and maps elements of various model views onto variation points. Thus, this approach realizes an *annotation-wise external* mapping and requires to assign annotations once manually by adding the elements to the corresponding annotation. More specifically, **VaVe** [1] is an extension to the multi-view framework **Vitruvius** [31] to realize a delta-oriented product line development. In the VaVe metamodel, a feature (called variant) is related with its implementation versions. Since for each variation point multiple versions may be present, it is an external annotation-wise mapping. Due to connecting multiple models by a correspondence model, a change in one model or of an annotation propagates automatically to corresponding model elements of other models. Thus, while an automated determination of the initial annotation is not regarded in both approaches, the annotations of multiple models are consistent *by construction*.

To sum it up, the way mappings are maintained in different approaches varies. Most notably, however, only one approach propagates annotations and none of the element-wise mapping approaches can ensure annotation consistency across models.

4.2 Model Transformation Approaches

Techniques to propagate annotations, independent of a concrete tool realization, are so called *multi-variant model transformations* [19, 50, 51] which typically extend single-variant (i.e., state-of-the-art) model transformation. Westfechtel and Greiner [51] discuss the differences of various solutions propagating annotations. To keep this article self-contained, here we discuss key techniques with focus on their degree of automation and genericity which is relevant to be integrable into MDPLE tools, for instance, those described above. Please note: variability rules [45, 47, 48] do not propagate annotations but vary the transformation itself to generate different products.

4.2.1 Transformation Classification Criteria. Figure 5 organizes classification criteria for multi-variant model transformations in a feature model: The *scope* describes whether the transformation propagation solution is specific to a transformation *definition*, particularly for its corresponding metamodels or for a transformation *language*. A solution working independently of the definition and the language resides at the highest level of genericity. Moreover, the level of *abstraction* implies whether the propagation realization requires modifications of existing transformation languages or their execution environments (engine). While *white-box* solutions either directly interact with the transformation definition (e.g., by analyzing it) or the engine (e.g., by changing its semantics), black-box solutions do not interact with the definition or the execution engine, thus, are realized in a *noninvasive* way. A *gray-box* solution requires some artifacts of the transformation which allow to draw conclusions on the definition, such as a trace, but it does not manipulate the definition nor the execution. Lastly, the approaches vary at the point in time at which annotations are propagated, which is either during the execution of the reused transformation (*inter-processing*) or afterwards (*post-processing*). On the whole, for integrating a solution into an existing tool a high degree of genericity, making it broadly applicable to various scenarios, as well as the realization and abstraction criteria are essential. It is easier to integrate a generic solution which does not require further modifications (noninvasive).

4.2.2 Multi-Variant Model Transformations. Key techniques, summarized in Table 2, are the *lifting* [38] of model transformations, a *propagation DSL* [9], *aspect-oriented* [21] propagation in the language Xpand [30], *trace-based* [50] propagation based on generic traces written by an arbitrary transformation language and a *hybrid* [19] solution of exploiting traces and analyzing an execution model of the ATL EMF/TVM [49] transformation specifications.

Lifting is a white-box solution defined for graph transformations. It requires to modify the execution semantics of state-of-the-art graph transformations to propagate annotations during the execution. Although the scope is restricted and the engine needs to be adapted, the solution is formally proven to satisfy commutativity.

The aspect-oriented solution offers a generic Xpand aspect which is triggered for each source model element to assign an annotation to the target element in case the source model element is annotated. This inter-processing solution, which works in a noninvasive way, is specific to the language Xpand but independent of the transformation definition. Since all capabilities of Xpand are reused without modification it is a black-box solution.

A post-processing black-box approach which is also performed in a noninvasive way offers a DSL with which correspondences of source and target elements of two metamodels can be expressed. Based on this script, annotations are propagated automatically. Even though the solution is language-independent, it requires to specify a correspondence script for each new transformation definition.

Lastly, the hybrid solution works for ATL/EMFTVM transformations only (*language-specific*) by exploiting trace information as well as by analyzing the execution model (*white-box*). This post-processing solution is generalized as a trace-based solution which works without analyzing language-specific bytecode and, thus, is a language- and definition independent gray-box solution, which is proven to satisfy commutativity for its computational model.

Although all approaches propagate annotations automatically, they reside at different levels of scope and abstraction. While lifting and the trace-based propagation are proven to behave correctly, the trace-based propagation may support easier integration and a broader set of transformation scenarios due to its language- and definition-independent design.

Table 2: Classification of multi-variant model transformations.

Ref	description	scope	abstraction	realization	time
[38]	lifting	definition-independent, language-specific	white-box	invasive, engine	inter
[20]	aspect-oriented	definition-independent, language-specific	black-box	noninvasive	inter
[9]	DSL	definition-specific, language-independent	black-box	noninvasive	post
[50, 51]	trace-based	definition-independent, language-independent	gray-box	noninvasive	post
[19]	hybrid	definition-independent, language-specific	white-box	noninvasive	post

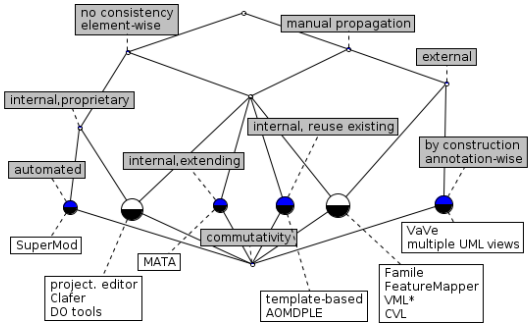

Figure 6: Formal Concept Analysis of the tools w.r.t. to the classification criteria.

4.3 Discussion

Summing it up, on the one hand, the classification of the tools reveals that the placement of the annotations varies but only one tool propagates annotations and only multi-view approaches natively ensure annotation consistency across models. On the other hand, several automated transformation techniques exist, some of them designed in an entirely generic way, to solve exactly this task. As demonstrated in the motivating example, propagating annotations automatically is indispensable to reduce the laborious and error-prone manual annotation process, particularly in annotative approaches. Accordingly, we answer the research questions as follows.

R1: As visualized in Figure 6 in form of a *formal concept analysis*, the maintenance of annotations in single models is manifold. Annotations are either stored externally or internally, the latter frequently by employing a proprietary or reusing an existing language. Furthermore, the most frequent form are element-wise mappings.

R2: Except for SuperMod, all of the MDPLE approaches miss an automated propagation of annotations. In addition, only multi-view approaches guarantee annotation consistency across models by construction. Contrastingly, multi-variant model transformations solve *both* problems.

Consequently, while the ways of storing annotations are manifold, almost all of them are maintained manually, yielding the need to integrate automated approaches for their maintenance.

As threats to validity we identify the selection of MDPLE tools. The set of MDPLE approaches is not exhaustive including every tool and approach which has been published once but comprises tools and languages covering each of the three variability mechanisms. A broader landscape may also give rise to refinements of the classification criteria for maintaining annotations.

5 RELATED WORK

This section presents related work on classifying annotation maintenance in (multiple) product line models.

An extensive literature review examines end-to-end traceability [29]. While this work studies the realization of traceability between artifacts at various development stages, it does not investigate particularly *how* annotations are assigned to model elements and preserved across different models which is the focus of our work.

Schwägerl and Westfechtel [42] compare ways to combine the three disciplines, MDSE, SPLE and software configuration management. As one combination, they classify MDPLE approaches based on the criteria of positive and negative variability. Our classification goes beyond this work and considers different forms of mappings and how annotations are maintained across different models.

Moreover, we introduce multi-variant model transformation techniques. A detailed discussion can be found in our recent work [51]. The main insights are included here since these transformations are a key technology to automatically propagate annotations.

6 CONCLUSION AND FUTURE WORK

In summary, we contribute classification criteria for model-driven product line approaches which allow for maintaining the annotations of multiple models. Our findings reveal, that there is a broad landscape of tools and approaches which rarely consider an automated strategy to maintain the annotation of multiple models, although transformation approaches have been developed to automate exactly that process and even prove its correctness.

Since the present classification does not cover the entire landscape of model-driven product line approaches, it can be extended by a systematic literature review which may reveal further details to classify particularly the maintenance of annotations across models.

REFERENCES

[1] Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf H. Reussner. 2018. Variants and Versions Management for Models with Integrated Consistency Preservation. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018*, Rafael Capilla, Malte Lochau, and Lidia Fuentes (Eds.). ACM, 3–10. https://doi.org/10.1145/3168365.3168377

[2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer. https://doi.org/10.1007/978-3-642-37521-7

[3] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The*

Netherlands, October 12-13, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6563), Brian A. Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer, Eindhoven, Netherlands, 102–122. https://doi.org/10.1007/978-3-642-19440-5_7

[4] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2016. Clafer: unifying class and feature modeling. *Software and Systems Modeling* 15, 3 (2016), 811–845. https://doi.org/10.1007/s10270-014-0441-1

[5] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30, 6 (2004), 355–371. https://doi.org/10.1109/TSE.2004.23

[6] Kathrin Berg, Judith Bishop, and Dirk Muthig. 2005. Tracing Software Product Line Variability: From Problem to Solution Space *(SAICSIT '05)*. South African Institute for Computer Scientists and Information Technologists, ZAF, 182–191.

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice*. Vol. 3. Morgan & Claypool Publishers. 1–207 pages.

[8] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. 2010. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (Antwerp, Belgium) *(ASE '10)*. ACM, New York, NY, USA, 173–174. https://doi.org/10.1145/1858996.1859032

[9] Thomas Buchmann and Sandra Greiner. 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*. 326–335. https://doi.org/10.5220/0006563403260335

[10] Thomas Buchmann and Felix Schwägerl. 2012. FAMILE: Tool support for evolving model-driven product lines. In *Joint Proc. co-located Events at 8th ECMFA (CEUR WS)*. Lyngby, Denmark, 59–62.

[11] Paul Clements and Linda Northrop. 2005. *Software Product Lines: Practices and Patterns* (3 ed.). Addison-Wesley.

[12] Krzysztof Czarnecki and MichałAntkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. 4th GPCE*. Springer, Tallinn, Estonia, 422–437. https://doi.org/10.1007/11561347_28

[13] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. 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*. 126–127. https://doi.org/10.1145/1094855.1094896

[14] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying feature-based model templates against well-formedness OCL constraints. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen (Eds.). ACM, 211–220. https://doi.org/10.1145/1173706.1173738

[15] Jaime Font, Manuel Ballarín, Øystein Haugen, and Carlos Cetina. 2015. Automating the variability formalization of a model family by means of common variability language. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 411–418. https://doi.org/10.1145/2791060.2793678

[16] Hassan Gomaa. 2006. Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*. 218. https://doi.org/10.1109/SPLINE.2006.1691600

[17] Hassan Gomaa and Michael E. Shin. 2002. Multiple-View Meta-Modeling of Software Product Lines. In *8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), 2-4 December 2002, Greenbelt, MD, USA*. IEEE Computer Society, 238–246. https://doi.org/10.1109/ICECCS.2002.1181517

[18] Sandra Greiner, Thomas Buchmann, and Bernhard Westfechtel. 2016. Bidirectional Transformations with QVT-R: A Case Study in Round-trip Engineering UML Class Models and Java Source Code. In *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016*. 15–27. https://doi.org/10.5220/0005644700150027

[19] Sandra Greiner, Felix Schwägerl, and Bernhard Westfechtel. 2017. Realizing Multi-variant Model Transformations on Top of Reused ATL Specifications. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, Luis Ferreira Pires, Slimane Hammoudi, and Bran Selic (Eds.). SCITEPRESS Science and Technology Publications, Portugal, Porto, Portugal, 362–373. https://doi.org/10.5220/0006137803620373

[20] Sandra Greiner and Bernhard Westfechtel. 2018. Evaluating Multi-variant Model-To-Text Transformations Realized by Generic Aspects. In *Model-Driven Engineering and Software Development - 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers (Communications in Computer and Information Science, Vol. 991)*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic (Eds.). Springer, 82–105. https://doi.org/10.1007/978-3-030-11030-7_5

[21] Sandra Greiner and Bernhard Westfechtel. 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*. 36–47. https://doi.org/10.5220/0006536700360047

[22] Sandra Greiner and Bernhard Westfechtel. 2020. Towards iterative software product line engineering with incremental multi-variant model transformations. In *VaMoS '20: 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg Germany, February 5-7, 2020*, Maxime Cordy, Mathieu Acher, Danilo Beuche, and Gunter Saake (Eds.). ACM, 22:1–22:9. https://doi.org/10.1145/3377024.3377032

[23] Iris Groher and Markus Völter. 2009. Aspect-Oriented Model-Driven Software Product Line Engineering. *LNCS Trans. Aspect Oriented Softw. Dev.* 6 (2009), 111–152. https://doi.org/10.1007/978-3-642-03764-1_4

[24] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *Proc. 12th SPLC*. Limerick, Ireland, 139–148. https://doi.org/10.1109/SPLC.2008.25

[25] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. 2009. Closing the Gap between Modelling and Java. In *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5969)*, Mark van den Brand, Dragan Gasevic, and Jeff Gray (Eds.). Springer, 374–383. https://doi.org/10.1007/978-3-642-12107-4_25

[26] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: Mapping features to models. In *Companion Proc. 30th ICSE*. ACM, Leipzig, Germany, 943–944.

[27] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2019. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *Art Sci. Eng. Program.* 3, 1 (2019), 2. https://doi.org/10.22152/programming-journal.org/2019/3/2

[28] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie-Mellon University, Software Engineering Institute.

[29] Jingyu Kim, Sungwon Kang, and Jihyun Lee. 2014. A Comparison of Software Product Line Traceability Approaches from End-to-End Traceability Perspectives. *International Journal of Software Engineering and Knowledge Engineering* 24, 04 (2014), 677–714. https://doi.org/10.1142/S0218194014500260

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

[31] Max E. Kramer, Erik Burger, and Michael Langhammer. 2013. View-Centric Engineering with Synchronized Heterogeneous Models. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling* (Montpellier, France) *(VAO '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2489861.2489864

[32] Michael Nieke, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: an integrated tool suite for modeling evolving context-aware software product lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017*, Maurice H. ter Beek, Norbert Siegmund, and Ina Schaefer (Eds.). ACM, 92–99. https://doi.org/10.1145/3023956.3023962

[33] Object Management Group (OMG). 2017. *Unified Modeling Language* (formal/2017-12-05 ed.). Needham, MA. https://www.omg.org/spec/UML/2.5.1/PDF.

[34] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL – A Delta-Based Modeling Framework for Software Product Line Engineering. In *International Conference on Automated Software Engineering (ASE)*. IEEE, Lincoln, NE, USA, 852–857. https://doi.org/10.1109/ASE.2015.106

[35] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3336294.3336299

[36] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Germany.

[37] Dennis Reuling, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2020. Towards projectional editing for model-based SPLs. In *VaMoS '20: 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg Germany, February 5-7, 2020*, Maxime Cordy, Mathieu Acher, Danilo Beuche, and Gunter Saake (Eds.). ACM, 25:1–25:10. https://doi.org/10.1145/3377024.3377030

[38] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting model transformations to product lines. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 117–128. https://doi.org/10.1145/2568225.2568267

[39] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6287)*, Jan Bosch and Jaejoon Lee (Eds.). Springer, 77–91.

https://doi.org/10.1007/978-3-642-15579-6_6

[40] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proceedings (ACM International Conference Proceedings Series)*, Patrick Heymans, Krzysztof Czarnecki, and Ulrich W. Eisenecker (Eds.). ACM, 119–126. https://doi.org/10.1145/1944892.1944907

[41] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: tool support for collaborative filtered model-driven software product line engineering. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 822–827. https://doi.org/10.1145/2970276.2970288

[42] Felix Schwägerl and Bernhard Westfechtel. 2017. Perspectives on combining model-driven engineering, software product line engineering, and version control. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017*, Maurice H. ter Beek, Norbert Siegmund, and Ina Schaefer (Eds.). ACM, 76–83. https://doi.org/10.1145/3023956.3023969

[43] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore - A Model-Based Delta Language Generation Framework. In *Modellierung 2014, 19.-21. März 2014, Wien, Österreich (LNI, Vol. P-225)*, Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer (Eds.). GI, 81–96. https://dl.gi.de/20.500.12116/17067

[44] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Integrated Management of Variability in Space and Time in Software Families. In *International Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/2648511.2648514

[45] Marten Sijtema. 2010. Introducing variability rules in ATL for managing variability in MDE-based product lines. *Proc. of MtATL* 10 (2010), 39–49.

[46] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. 2006. *Model-driven software development - technology, engineering, management*. Pitman.

[47] Daniel Strüber, Sven Peldszus, and Jan Jürjens. 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*. 337–355. https://doi.org/10.1007/978-3-319-89363-1_19

[48] Daniel Strüber and Stefan Schulz. 2016. A Tool Environment for Managing Families of Model Transformation Rules. In *Graph Transformation - 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings*. 89–101. https://doi.org/10.1007/978-3-319-40530-8_6

[49] Dennis Wagelaar, Ludovico Iovino, Davide Di Ruscio, and Alfonso Pierantonio. 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*. 192–207. https://doi.org/10.1007/978-3-642-30476-7_13

[50] Bernhard Westfechtel and Sandra Greiner. 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*. 46–56. https://doi.org/10.1145/3239372.3239414

[51] Bernhard Westfechtel and Sandra Greiner. 2020. Extending single- to multi-variant model transformations by trace-based propagation of variability annotations. *Software & Systems Modeling* 19, 4 (2020), 853–888. https://doi.org/10.1007/s10270-020-00791-9

[52] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. 2009. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *LNCS Transactions on Aspect Oriented Software Development* VI (2009), 191–237. https://doi.org/10.1007/978-3-642-03764-1_6

[53] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. 2011. A Survey on UML-based Aspect-oriented Design Modeling. *Comput. Surveys* 43, 4 (Oct. 2011), 28:1–28:33.

[54] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. 2003. Towards a UML Profile for Software Product Lines. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers (Lecture Notes in Computer Science, Vol. 3014)*, Frank van der Linden (Ed.). Springer, 129–139. https://doi.org/10.1007/978-3-540-24667-1_10

[55] Steffen Zschaler, Pablo Sánchez, and João Santos. 2010. VML* — A Family of Languages for Variability Management in Software Product Lines. In *Proc. 3rd SLE*. Springer, Denver, CO, USA, 82–102.